Detection of Unnecessary Dependencies in C Programs

Atalay Pabuşçu
dept. Computer Science
Ozyegin University
Istanbul, Turkey
atalay.pabuscu@ozu.edu.tr

***Abstract*— In this project, we aim to find unnecessary dependencies in C programs especially used in embedded systems. In big companies, a group of software developers work together on the same projects and generally developed programs get enhance inherited. This project's goal is to find a way to detecting unnecessary dependencies such as unused functions, variables and included header files.**

***Index Terms*—Software Architecture, GNU cflow, ReGex, embedded systems, header files, C language optimization,**

I. Introduction

In this project, we focus on the problem of unnecessary dependency usage in C codes. The main goal of this project is optimizing an Embedded System software. The Embedded System developers encounter with different types of limitations (code flash, data flash, power consumption, oscillator frequency) in software developing phase due to companies they work for prefer low cost microcontrollers. We developed an executable program that working on Linux OS. This program finds and returns warning for unnecessary dependencies in stated project files. The program executes generically thus it can be used on any C projects. For some reasons (stack size, code size) compiled programs can return an error during linker process due to flash/ram size of microcontroller in big sized projects. This project may give alternative solution to overcome this type of problems.

II. Background

A. Methodology

In order to actualize our purpose we firstly applied number of parsing operations project source and header files and extract any information about project (header file names, number of dependencies of each header file, most included header files etc.). In addition to warn any unnecessary file dependencies, we provided an analyze of given project structure to developers. Because it is extremely useful for the developer to know which files are depends on investigated file or vice versa. Then we used flow graph generator tool that works with C source files, for detecting all defined but never used global functions. As a result of applying this methodology, we observed the program output was really useful for decreasing code size and decreasing compilation time in C codes that used in Embedded Systems.

B. Technique

We used the Python programming language in order to extract all project file tree, all included header files and functions. We will exploit from regular expressions (ReGex) to search patterns (definitions) in source/header files [1]. We used GNU cflow tool in order to find function dependencies [5]. The outputs of our used cflow commands, contain information references of used functions, defined functions, called (invoked) functions. We parsed the content of these outputs and we detected functions that defined and declared globally but never used in stated project. We used E2 Studio (for Renesas RX MCU projects), IAR Embedded Workbench (for Renesas RL projects), and ST Visual Develop (for ST MCU projects) IDEs in order to evaluate the given embedded code in terms of optimization (code size, compilation time). We evaluated the code before and after applying directives of our program (Removing unnecessary header file inclusions and unused functions).

III. The Project Work

In our work, we generally focused on parsing strings that located in project source/header files and cflow outputs, by using ReGex in order to optimize the C codes that used in embedded systems. We evaluated our works using the embedded system projects that developed in the R&D of a large white goods manufacturer company. Since sharing and distribution of the code is forbidden, we will only show small excerpts from the projects. Later, we will show the changes in compile time and code size separately for each project in Results & Discussion section.

A. Detecting Unnecessary File Dependency

Before explaining what we did in this step of our project, we would like to give information about how the program we developed works. Our program is an executable file that runs on Linux operating systems. We used Ubuntu v20.04.4 (LTS) as operating system in order to develop and execute our program. It is necessary to give the directory of the project file to be analyzed in our program. Or you can just put our program and project folder in the same folder. When our program runs, it automatically understands that there is only one project folder in the given directory and starts to examine the project file. If the program finds more than one project folder in the given directory, it returns an error message to the user. The possible use of our program is shown in Figure 1.

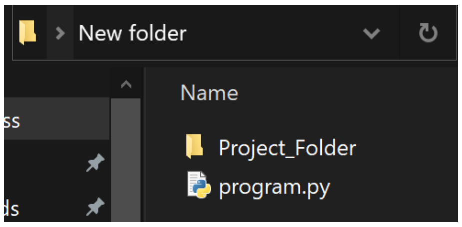


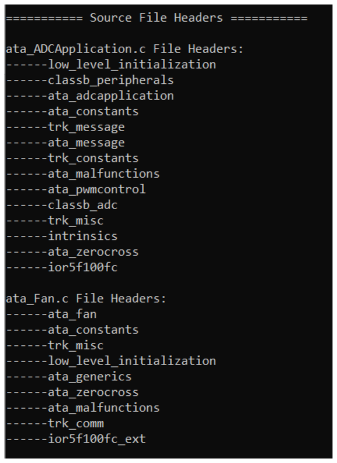
Figure 1: The possible use of our developed program (Put the program and project folder into same folder)

When the program runs, it first shows the user the folder tree containing all the files of the project. The project files given to the program may not always contain only C source files and Header files. However, code that runs in embedded systems (in microcontroller) can only be created (object file) with C source files and Header files or an optimization process can be made only through these files. Therefore, the program continues to work by filtering only C source files and Header files among all files. The extensions required for this filtering process are ".c" and ".h" extensions. The ReGex expressions that used to find only these files are as follows [2].

```
# Set Project Source and Header Files
    using ".c" and ".h" file extensions
def
    app_SetProjectSourceAndHeaderFiles(path,
    regex):
   ...
   if re.findall("\.c$", str(file)):
      source_files_loc.append(path +
          "/" + file)
      file = file[:(re.search('\.c$',
          file).start())]
      source_files.append(file)
   if re.findall("\.h$", str(file)):
      file = file[:(re.search('\.h$',
          file).start())]
      if file.upper() not in
          list(header_files.keys()): #
          Prevent same file names in
          search tree
      header_files_loc.append(path +
          "/" + file + ".h")
      header_files[file.upper()] = []
          # We do not want to header
          file research case sensitive
```

After our program finds the necessary files, it starts the text parsing process in the files with the help of ReGex. The aim of the project at this stage is to find all included header files in all files (source/header files). These header files detected using Python language and ReGex are stored in a dictionary variable. The reason we use the dictionary variable is to store the other included header files in the found header files. Afterwards, the program shows the user all header files included to C source files and Header files as in Figure 2. A small part of the Python function that allows us to find the header files is given below.

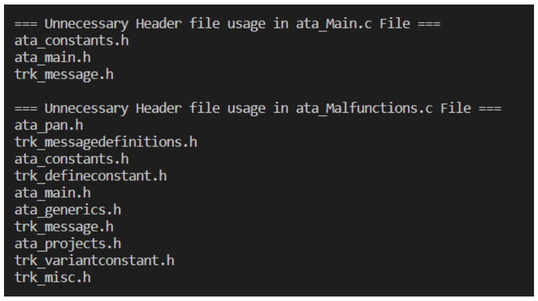
```
# Find and get all header files in
    given file (parameter: file path)
def app_GetHeadersFromFile(path):
   ...
   if ("#include" in line_str) and
       (re.findall('\.h[>|"]$',
      str(line_str))):
      temp_line =
          line_str[re.search('#include
          .*?[<|"]', line_str).end():]
      temp_line =
          temp_line[:(re.search('\.h[>|"]',
          temp_line).start())]
      headers.append(temp_line.upper())
```



```
=========== Source File Headers ===========

ata_ADCApplication.c File Headers:
------low_level_initialization
------classb_peripherals
------ata_adcapplication
------ata_constants
------trk_message
------ata_message
------trk_constants
------ata_malfunctions
------ata_pwmcontrol
------classb_adc
------trk_misc
------intrinsics
------ata_zerocross
------ior5f100fc

ata_Fan.c File Headers:
------ata_fan
------ata_constants
------trk_misc
------low_level_initialization
------ata_generics
------ata_zerocross
------ata_malfunctions
------trk_comm
------ior5f100fc_ext
```

Figure 2: The possible use of our developed program (Put the program and project folder into same folder)

The main purpose of our project is to find unnecessary dependencies and warn the developer about it. For this, we showed the user unnecessary header files in this step. The output of the program for a sample project is as in Figure 3. The basic logic of finding unnecessary files is to find other header files in the header file that are included by source code file and compare these header files with the other header files that are included in the source code.

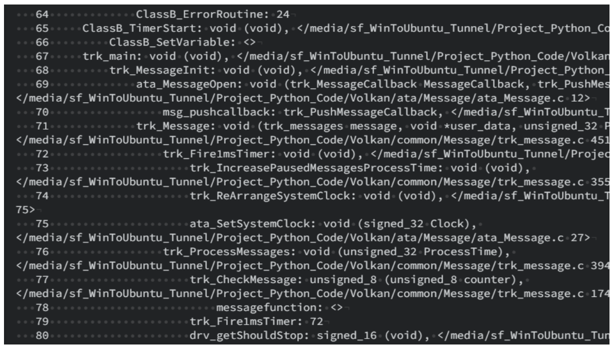


```
=== Unnecessary Header file usage in ata_Main.c File ===
ata_constants.h
ata_main.h
trk_message.h

=== Unnecessary Header file usage in ata_Malfunctions.c File ===
ata_pan.h
trk_messagedefinitions.h
ata_constants.h
trk_defineconstant.h
ata_main.h
ata_generics.h
trk_message.h
ata_projects.h
trk_variantconstant.h
trk_misc.h
```

Figure 3: Program output that shows unnecessary file dependency for a sample project

B. Detecting Unused Global Functions

The main purpose of our project at this stage is to inform/warn the user of global functions that are declared (extern) in a header file and defined in a C source file, but not used anywhere in the project. For this, we decided to use the GNU cflow tool mentioned in the Technique section. GNU cflow is a software tool that examines function dependencies of all kinds of C and Cpp source files that can be compiled with the gcc compiler on Linux operating systems. It can produce different types of dependency flow outputs. Generally, Direct, Reverse and POSIX format outputs are used [6].

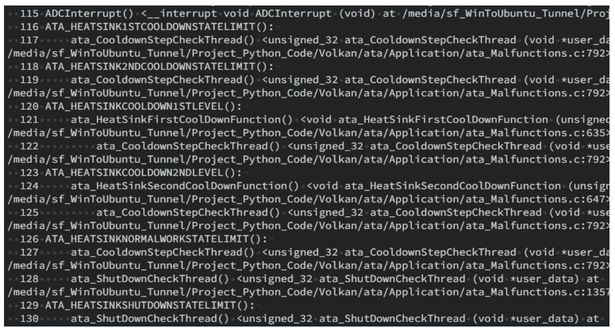


```
   64             ClassB_ErrorRoutine: 24
   65     ClassB_TimerStart: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_Co
   66         ClassB_SetVariable: <>
   67     trk_main: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan
   68         trk_MessageInit: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_
   69             ata_MessageOpen: void (trk_MessageCallback MessageCallback, trk_PushMes
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Message/ata_Message.c 12>
   70                 msg_pushcallback: trk_PushMessageCallback, </media/sf_WinToUbuntu_T
   71             trk_Message: void (trk_messages message, void *user_data, unsigned_32 P
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 451
   72                 trk_Fire1msTimer: void (void), </media/sf_WinToUbuntu_Tunnel/Projec
   73                     trk_IncreasePausedMessagesProcessTime: void (void),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 355
   74                     trk_ReArrangeSystemClock: void (void), </media/sf_WinToUbuntu_T
75>
   75                     ata_SetSystemClock: void (signed_32 Clock),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Message/ata_Message.c 27>
   76                 trk_ProcessMessages: void (unsigned_32 ProcessTime),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 394
   77                     trk_CheckMessage: unsigned_8 (unsigned_8 counter),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 174
   78                         messagefunction: <>
   79                     trk_Fire1msTimer: 72
   80                     drv_getShouldStop: signed_16 (void), </media/sf_WinToUbuntu_Tun
```

Figure 4: GNU cflow POSIX format output for a sample project

In our project, we worked with output in two different formats: Reverse and POSIX format. We experimented with both for a long time and successfully mapped the function dependency of the program. Direct format starts from the "main" function of a C program and outputs other functions that are called directly inside the functions, the parameters they take, the data types they return, and the files they are called from. The POSIX format is very similar to the Direct format. The difference is that each line of the output has a reference number. It also uses "<>" signs instead of parentheses when giving invoked functions. Therefore, it is not confused with the opening parentheses for the function parameters. This makes the POSIX format a more parseable format. A cflow output in POSIX format is shown in Figure 4.

Another output format we use is the Reverse format. In this format, the functions from which they are called (reference) are output, not the functions they call. So, reverse research is being done. This makes the Reverse cflow format the most usable cflow output format for our purpose. Because the purpose of the project at this stage is to find unused global functions. A cflow output in reverse format is shown in Figure 5.



```
  115 ADCInterrupt() <__interrupt void ADCInterrupt (void) at /media/sf_WinToUbuntu_Tunnel/Proj
  116 ATA_HEATSINK1STCOOLDOWNSTATELIMIT():
  117     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792>
  118 ATA_HEATSINK2NDCOOLDOWNSTATELIMIT():
  119     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792>
  120 ATA_HEATSINKCOOLDOWN1STLEVEL():
  121     ata_HeatSinkFirstCoolDownFunction() <void ata_HeatSinkFirstCoolDownFunction (unsigned
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:635>
  122         ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *use
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792>
  123 ATA_HEATSINKCOOLDOWN2NDLEVEL():
  124     ata_HeatSinkSecondCoolDownFunction() <void ata_HeatSinkSecondCoolDownFunction (unsign
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:647>
  125         ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *use
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792>
  126 ATA_HEATSINKNORMALWORKSTATELIMIT():
  127     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792>
  128     ata_ShutDownCheckThread() <unsigned_32 ata_ShutDownCheckThread (void *user_data) at
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:1357
  129 ATA_HEATSINKSHUTDOWNSTATELIMIT():
  130     ata_ShutDownCheckThread() <unsigned_32 ata_ShutDownCheckThread (void *user_data) at
```

Figure 5: GNU cflow Reverse format output for a sample project

We wrote different GNU cflow linux commands in Python script for generating different format types of cflow outputs.

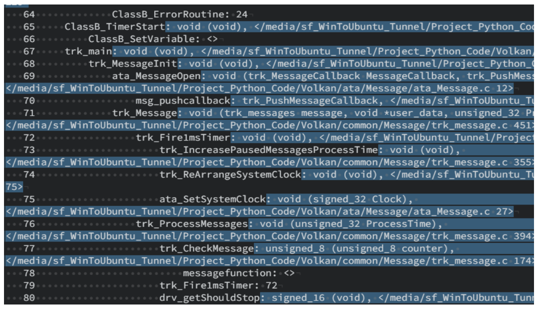
```
CMD_CFLOW_POSIX = \
"cflow --number --brief --format=posix"
CMD_CFLOW_RVRS =
"cflow --number --brief --reverse"
CMD_CFLOW_BASIC = \
"cflow --number -d2 --reverse"
```

First command generates POSIX format output, second command generates Reverse format output and the last command generates Reverse output with using maximum two function references property. The third command gives less information about function dependencies.

All the string decomposition processes we applied using ReGex for a cflow output in POSIX format are presented in the following images, respectively.Here is an excerpt from the string parsing method applied for the POSIX format.

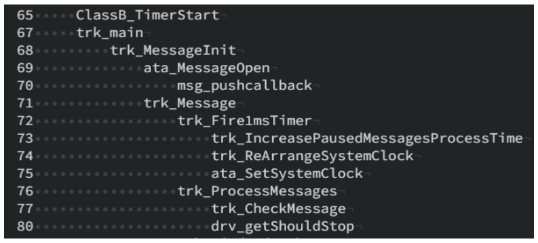
```
# GNU cflow POSIX format output
    parsing using ReGex
...
def_rgx_rule = ": .*, <.*"
# cflow --format=posix
if cflow_format == CFLOW_FORMAT_POSIX:
   if re.findall(def_rgx_rule,
       str(line_str)):
      temp_line =
          line_str[:re.search(def_rgx_rule,
          line_str).start()]
```

```
if re.findall("([ ]{2,})" ,
    temp_line): # If whitespace
    is less than 2, it shows just
    definition of the function
     (Not used in this scope)
   temp_line =
       temp_line[((re.search(\
   "\d.*?[a-zA-Z]",
       temp_line).end())-1):]
   called_funcs.add(temp_line) #
       Add functions to global
       called function list
else:
   temp_line =
       temp_line[((re.search(\
   "\d.*?[a-zA-Z]",
       temp_line).end())-1):]
   defined_funcs.append(temp_line)
```



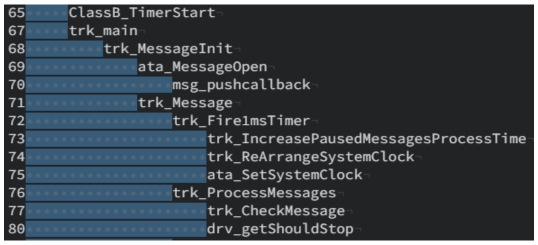
```
64              ClassB_ErrorRoutine: 24
65      ClassB_TimerStart: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_Coc
66          ClassB_SetVariable: <>
67      trk_main: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/
68          trk_MessageInit: void (void), </media/sf_WinToUbuntu_Tunnel/Project_Python_C
69              ata_MessageOpen: void (trk_MessageCallback MessageCallback, trk_PushMess
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Message/ata_Message.c 12>
70                  msg_pushcallback: trk_PushMessageCallback, </media/sf_WinToUbuntu_Tu
71              trk_Message: void (trk_messages message, void *user_data, unsigned_32 Pr
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 451>
72                  trk_Fire1msTimer: void (void), </media/sf_WinToUbuntu_Tunnel/Project
73                      trk_IncreasePausedMessagesProcessTime: void (void),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 355>
74                      trk_ReArrangeSystemClock: void (void), </media/sf_WinToUbuntu_Tu
75>
75                      ata_SetSystemClock: void (signed_32 Clock),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Message/ata_Message.c 27>
76                  trk_ProcessMessages: void (unsigned_32 ProcessTime),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 394>
77                      trk_CheckMessage: unsigned_8 (unsigned_8 counter),
</media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/common/Message/trk_message.c 174>
78                          messagefunction: <>
79                      trk_Fire1msTimer: 72
80                      drv_getShouldStop: signed_16 (void), </media/sf_WinToUbuntu_Tunn
```

Figure 6: GNU cflow POSIX format output parsing - ReGex Array-0



```
65    ClassB_TimerStart
67    trk_main
68        trk_MessageInit
69            ata_MessageOpen
70                msg_pushcallback
71            trk_Message
72                trk_Fire1msTimer
73                    trk_IncreasePausedMessagesProcessTime
74                    trk_ReArrangeSystemClock
75                    ata_SetSystemClock
76                trk_ProcessMessages
77                    trk_CheckMessage
80                    drv_getShouldStop
```

Figure 7: GNU cflow POSIX format output parsing - ReGex Array-1

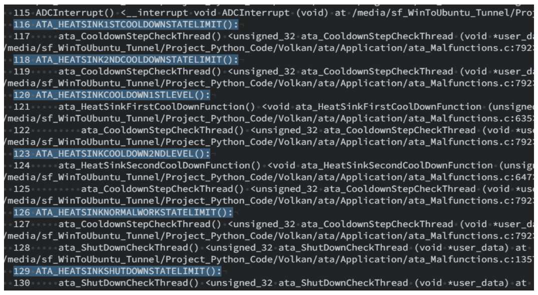


```
65    ClassB_TimerStart
67    trk_main
68        trk_MessageInit
69            ata_MessageOpen
70                msg_pushcallback
71            trk_Message
72                trk_Fire1msTimer
73                    trk_IncreasePausedMessagesProcessTime
74                    trk_ReArrangeSystemClock
75                    ata_SetSystemClock
76                trk_ProcessMessages
77                    trk_CheckMessage
80                    drv_getShouldStop
```

Figure 8: GNU cflow POSIX format output parsing - ReGex Array-2

All the string parsing processes we applied using ReGex for a cflow output in Reverse format are presented in the following images, in order. An excerpt from the string parsing method applied for the Reverse format is given below.

```
# GNU cflow Reverse format output
    parsing using ReGex
...
# cflow --reverse
   elif cflow_format ==
       CFLOW_FORMAT_RVRS:
      if re.findall("\d.*\(\):",
          str(line_str)):
         temp_line =
             line_str[((re.search(\
         "\d.*?[a-zA-Z]",
             line_str).end())-1):]
         temp_line =
             temp_line[:re.search("\(\).*",
             temp_line).start()]
         tmp_called_funcs.add(temp_line)
         called_funcs.add(temp_line) #
             Add functions to global
             called function list
      elif re.findall("\d.*\(\)(:|
          |$).*:$", str(line_str)):
         temp_line =
             line_str[((re.search(\
         "\d.*?[a-zA-Z]",
             line_str).end())-1):]
         temp_line =
             temp_line[:re.search("\(\).*",
             temp_line).start()]
         tmp_called_funcs.add(temp_line)
         called_funcs.add(temp_line) #
             Add functions to global
             called function list
```

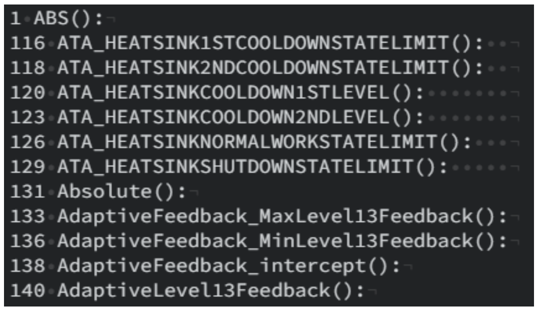


```
115 ADCInterrupt() <__interrupt void ADCInterrupt (void) at /media/sf_WinToUbuntu_Tunnel/Proj
116 ATA_HEATSINK1STCOOLDOWNSTATELIMIT():
117     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792
118 ATA_HEATSINK2NDCOOLDOWNSTATELIMIT():
119     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792
120 ATA_HEATSINKCOOLDOWN1STLEVEL():
121     ata_HeatSinkFirstCoolDownFunction() <void ata_HeatSinkFirstCoolDownFunction (unsigne
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:635
122         ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *use
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792
123 ATA_HEATSINKCOOLDOWN2NDLEVEL():
124     ata_HeatSinkSecondCoolDownFunction() <void ata_HeatSinkSecondCoolDownFunction (unsig
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:647
125         ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *use
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792
126 ATA_HEATSINKNORMALWORKSTATELIMIT():
127     ata_CooldownStepCheckThread() <unsigned_32 ata_CooldownStepCheckThread (void *user_da
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:792
128     ata_ShutDownCheckThread() <unsigned_32 ata_ShutDownCheckThread (void *user_data) at
/media/sf_WinToUbuntu_Tunnel/Project_Python_Code/Volkan/ata/Application/ata_Malfunctions.c:135
129 ATA_HEATSINKSHUTDOWNSTATELIMIT():
130     ata_ShutDownCheckThread() <unsigned_32 ata_ShutDownCheckThread (void *user_data) at
```

Figure 9: GNU cflow Reverse format output parsing - ReGex Array-0

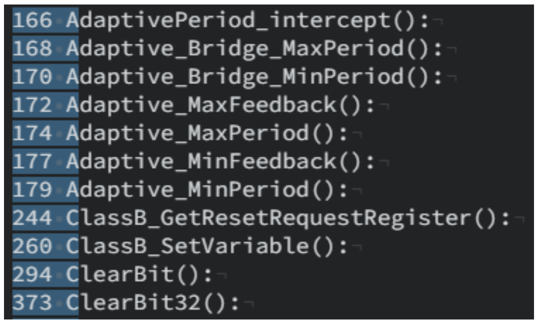
Using the GNU cflow ”function dependency extractor” tool and ReGex, we found all the used functions in a C project. We stored all the used function names in the ”set” variable to avoid name repetition.

In order to achieve the goal in this step of the project, we lastly needed to find the declarations of the global functions in the header files. It is impossible to find function dependencies in a C project using ReGex alone, but it is possible to find global functions using ReGex (from extern keyword). For this, we wrote a function that browses all header files and keeps all functions declared as extern in the



```
1 ABS():
116 ATA_HEATSINK1STCOOLDOWNSTATELIMIT():
118 ATA_HEATSINK2NDCOOLDOWNSTATELIMIT():
120 ATA_HEATSINKCOOLDOWN1STLEVEL():
123 ATA_HEATSINKCOOLDOWN2NDLEVEL():
126 ATA_HEATSINKNORMALWORKSTATELIMIT():
129 ATA_HEATSINKSHUTDOWNSTATELIMIT():
131 Absolute():
133 AdaptiveFeedback_MaxLevel13Feedback():
136 AdaptiveFeedback_MinLevel13Feedback():
138 AdaptiveFeedback_intercept():
140 AdaptiveLevel13Feedback():
```

Figure 10: GNU cflow Reverse format output parsing - ReGex Array-1

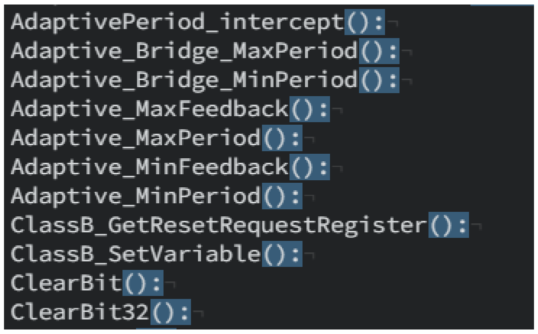


```
166 AdaptivePeriod_intercept():
168 Adaptive_Bridge_MaxPeriod():
170 Adaptive_Bridge_MinPeriod():
172 Adaptive_MaxFeedback():
174 Adaptive_MaxPeriod():
177 Adaptive_MinFeedback():
179 Adaptive_MinPeriod():
244 ClassB_GetResetRequestRegister():
260 ClassB_SetVariable():
294 ClearBit():
373 ClearBit32():
```

Figure 11: GNU cflow Reverse format output parsing - ReGex Array-2

"set" variable. A section of the function used is given below.

```
# Find all global functions that
    declared in a header file
extern_funcs = set({}) # Functions
    that declared global scope (We
    declare this variable as set due to
    prevent from duplication of
    function name)
...
rgx_rule = "^ *?extern .*\(.*\);"
...
if re.findall(rgx_rule, str(line_str)):
   temp_line =
       line_str[:re.search(".*\(",
       line_str).end()-1] # Add -1 in
```

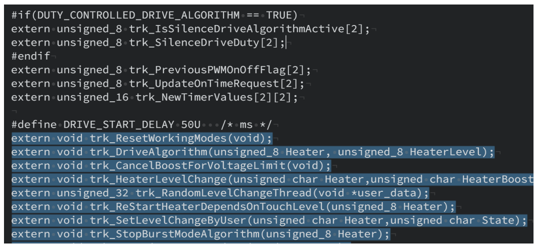


```
AdaptivePeriod_intercept():
Adaptive_Bridge_MaxPeriod():
Adaptive_Bridge_MinPeriod():
Adaptive_MaxFeedback():
Adaptive_MaxPeriod():
Adaptive_MinFeedback():
Adaptive_MinPeriod():
ClassB_GetResetRequestRegister():
ClassB_SetVariable():
ClearBit():
ClearBit32():
```

Figure 12: GNU cflow Reverse format output parsing - ReGex Array-3

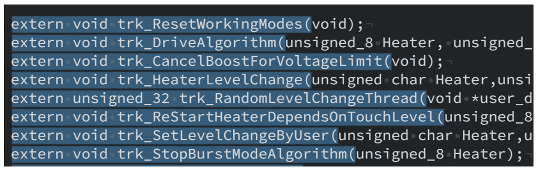
```
       order to ignore "(" paranthesis
   temp_line = temp_line[((re.search(\
   "extern.*?[a-zA-Z]",
       temp_line).end())-1):]
   temp_line =
       temp_line[(re.search(".* .*?",
       temp_line).end()):]
   ext_funcs.append(temp_line)
```

The matching results obtained with the applied ReGex expressions are given in the images below.



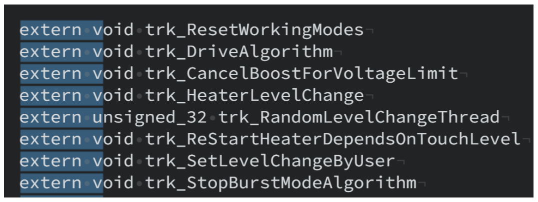
```
#if(DUTY_CONTROLLED_DRIVE_ALGORITHM == TRUE)
extern unsigned_8 trk_IsSilenceDriveAlgorithmActive[2];
extern unsigned_8 trk_SilenceDriveDuty[2];
#endif
extern unsigned_8 trk_PreviousPWMOnOffFlag[2];
extern unsigned_8 trk_UpdateOnTimeRequest[2];
extern unsigned_16 trk_NewTimerValues[2][2];

#define DRIVE_START_DELAY 50U   /* ms */
extern void trk_ResetWorkingModes(void);
extern void trk_DriveAlgorithm(unsigned_8 Heater, unsigned_8 HeaterLevel);
extern void trk_CancelBoostForVoltageLimit(void);
extern void trk_HeaterLevelChange(unsigned char Heater,unsigned char HeaterBoost
extern unsigned_32 trk_RandomLevelChangeThread(void *user_data);
extern void trk_ReStartHeaterDependsOnTouchLevel(unsigned_8 Heater);
extern void trk_SetLevelChangeByUser(unsigned char Heater,unsigned char State);
extern void trk_StopBurstModeAlgorithm(unsigned_8 Heater);
```

Figure 13: Matched strings for global function detection - ReGex Array-0



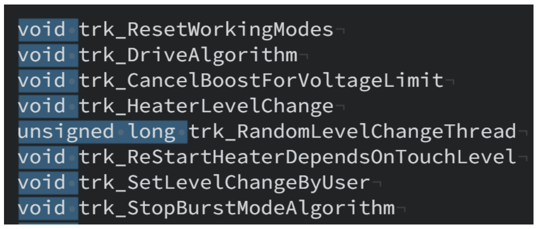
```
extern void trk_ResetWorkingModes(void);
extern void trk_DriveAlgorithm(unsigned_8 Heater, unsigned_
extern void trk_CancelBoostForVoltageLimit(void);
extern void trk_HeaterLevelChange(unsigned char Heater,unsi
extern unsigned_32 trk_RandomLevelChangeThread(void *user_d
extern void trk_ReStartHeaterDependsOnTouchLevel(unsigned_8
extern void trk_SetLevelChangeByUser(unsigned char Heater,u
extern void trk_StopBurstModeAlgorithm(unsigned_8 Heater);
```

Figure 14: Matched strings for global function detection - ReGex Array-1



```
extern void trk_ResetWorkingModes
extern void trk_DriveAlgorithm
extern void trk_CancelBoostForVoltageLimit
extern void trk_HeaterLevelChange
extern unsigned_32 trk_RandomLevelChangeThread
extern void trk_ReStartHeaterDependsOnTouchLevel
extern void trk_SetLevelChangeByUser
extern void trk_StopBurstModeAlgorithm
```

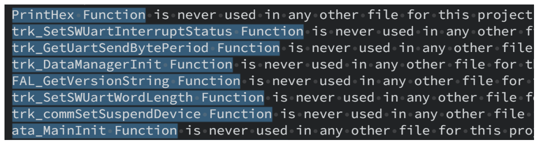
Figure 15: Matched strings for global function detection - ReGex Array-2



```
void trk_ResetWorkingModes
void trk_DriveAlgorithm
void trk_CancelBoostForVoltageLimit
void trk_HeaterLevelChange
unsigned long trk_RandomLevelChangeThread
void trk_ReStartHeaterDependsOnTouchLevel
void trk_SetLevelChangeByUser
void trk_StopBurstModeAlgorithm
```

Figure 16: Matched strings for global function detection - ReGex Array-3

As the last step of the program, all the "extern" functions that are found and all functions that are used (called) in the project are compared. All functions that are not called anywhere in the project but declared in a header file are given to the user as in Figure 17. It will be discussed in the Results & Discussion section whether removing all these unnecessary dependencies reported to the user from the project creates a performance increase and saves memory for the project.



```
PrintHex Function is never used in any other file for this project
trk_SetSWUartInterruptStatus Function is never used in any other f
trk_GetUartSendBytePeriod Function is never used in any other file
trk_DataManagerInit Function is never used in any other file for t
FAL_GetVersionString Function is never used in any other file for
trk_SetSWUartWordLength Function is never used in any other file f
trk_commSetSuspendDevice Function is never used in any other file
ata_MainInit Function is never used in any other file for this pro
```

Figure 17: Never used functions that are found by program for a sample project

IV. Results & Discussion

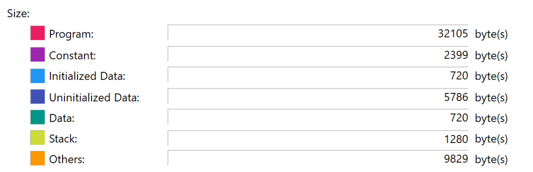
The main purpose of our project was to inform/warn the user about the unnecessary dependencies in the given project. Therefore, we do not do a refactoring on the project file. We only warn the user about these dependencies. We examined the change in performance and memory usage by removing all unnecessary dependencies that program we developed detects for 4 different white goods embedded system software projects. We will not share the source code of the projects directly, as all rights of the project's software are reserved, but we will show all the results that we obtained.

A. *Project-1*

The sample software used in this section belongs to an Oven embedded system project developed in the R&D department of a white goods company. In this project, the R5F51305 processor from the RX130 family produced by the Renesas company is used. All the results for this project were obtained from the IDE named E2 Studio, which was developed by the Renesas company for its own processors. We used a tool in the IDE that shows Memory Usage in detail with an interface. With this tool, we obtained information about used processor., such as program size, all the space occupied by constants, stack size. At the same time, the use of RAM and ROM resources is presented to the software developer in detail. The results obtained without following the directives given by the program we developed for Project-1 will be given in order.

In the output shown in Figure 18, it is seen that the size of the code embedded in the memory of the processor is 32105 bytes. The output also shows the allocated Stack size for all static and local variables (not dynamically generated) in the project.

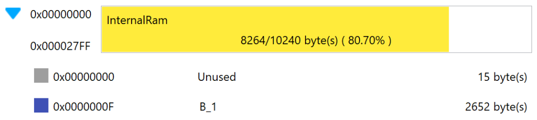
In Figure 19 and Figure 20, the allocation rate of the RAM and ROM areas of the microcontroller in the linker stage at the end of the compilation process is shown.



Size:

🟥 Program:	32105	byte(s)
🟪 Constant:	2399	byte(s)
🟦 Initialized Data:	720	byte(s)
🟦 Uninitialized Data:	5786	byte(s)
🟩 Data:	720	byte(s)
🟨 Stack:	1280	byte(s)
🟧 Others:	9829	byte(s)

Figure 18: Memory size of the Project-1 before program directives applied



0x00000000 – 0x000027FF	InternalRam	8264/10240 byte(s) (80.70%)
▨ 0x00000000	Unused	15 byte(s)
🟦 0x0000000F	B_1	2652 byte(s)

Figure 19: Internal RAM size of the Project-1 before program directives applied

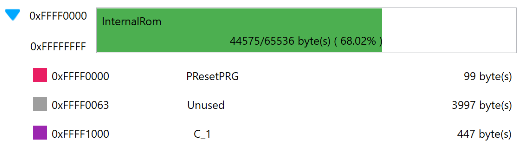
As shown in Figure 21, the total build time was measured as 26 seconds 987 milliseconds.

The results obtained after removing unnecessary dependencies from the project file by following the directives given by the program we developed for Project-1 are given below, respectively.

After following the directives given by the program we developed, it is seen in Figure 22 that the program size used for Project-1 is 31324 bytes. There was a 784 byte reduction in code size.

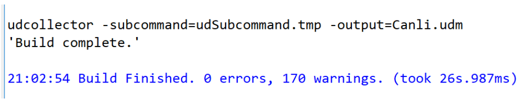
In Figure 23 and Figure 24, the allocation rate of the RAM and ROM areas of the microcontroller in the linker stage at the end of the compilation process is shown. Here, it was observed that the RAM usage decreased from 80.70% to 80.66%. ROM usage decreased from 68.02% to 66.81%. Since functions are stored in ROM and variables are stored in RAM, it is natural for the ROM section to be freed up more than RAM.

In Figure 25, It can be seen that the total compile/build time is 18 seconds and 683 milliseconds. We have observed that the compilation time is significantly reduced as a result of both removing un-



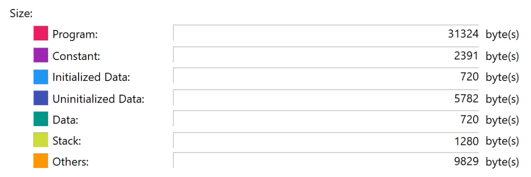
0xFFFF0000 – 0xFFFFFFFF	InternalRom	44575/65536 byte(s) (68.02%)
🟥 0xFFFF0000	PResetPRG	99 byte(s)
▨ 0xFFFF0063	Unused	3997 byte(s)
🟪 0xFFFF1000	C_1	447 byte(s)

Figure 20: Internal ROM size of the Project-1 before program directives applied



```
udcollector -subcommand=udSubcommand.tmp -output=Canli.udm
'Build complete.'

21:02:54 Build Finished. 0 errors, 170 warnings. (took 26s.987ms)
```

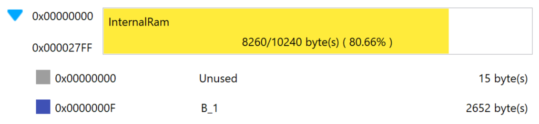
Figure 21: Build time of the Project-1 before program directives applied



Size:

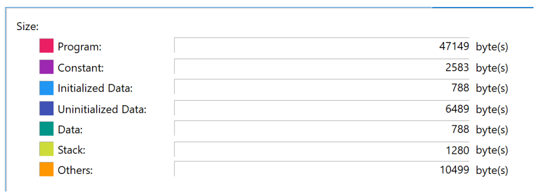
🟥 Program:	31324	byte(s)
🟪 Constant:	2391	byte(s)
🟦 Initialized Data:	720	byte(s)
🟦 Uninitialized Data:	5782	byte(s)
🟩 Data:	720	byte(s)
🟨 Stack:	1280	byte(s)
🟧 Others:	9829	byte(s)

Figure 22: Memory size of the Project-1 after program directives applied



```
0x00000000   InternalRam
0x000027FF   8260/10240 byte(s) ( 80.66% )
0x00000000   Unused    15 byte(s)
0x0000000F   B_1       2652 byte(s)
```

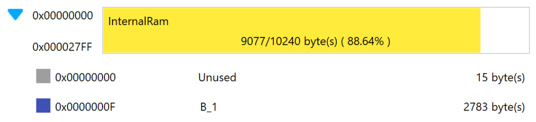
Figure 23: Internal RAM size of the Project-1 after program directives applied



Size:

🟥 Program:	47149	byte(s)
🟪 Constant:	2583	byte(s)
🟦 Initialized Data:	788	byte(s)
🟦 Uninitialized Data:	6489	byte(s)
🟩 Data:	788	byte(s)
🟨 Stack:	1280	byte(s)
🟧 Others:	10499	byte(s)

Figure 26: Memory size of the Project-2 before program directives applied



```
0x00000000   InternalRam
0x000027FF   9077/10240 byte(s) ( 88.64% )
0x00000000   Unused    15 byte(s)
0x0000000F   B_1       2783 byte(s)
```

Figure 27: Internal RAM size of the Project-2 before program directives applied

used but defined functions and removing unnecessary header file dependencies.

B. *Project-2*

The sample software used in this section belongs to an Induction Cooker embedded system project developed in the R&D department of a white goods company. In this project, the R5F51303ADFM processor from the RX130 family produced by the Renesas Company is used. All the results for this project are again obtained from the E2 Studio. The results obtained without following the directives given by the program we developed for Project-2 will be given in order.

We observed that the used program size was 47149 bytes as shown in Figure 26.

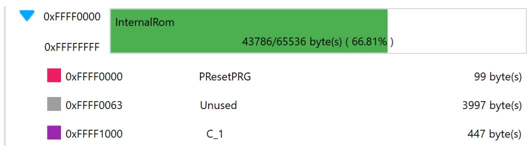
In Figure 27 and Figure 28, the allocation rate of the RAM and ROM areas of the microcontroller at the linker stage at the end of the compilation process is shown.

As it was seen in Figure 29, the total compilation time was measured as 22 seconds and 380 milliseconds. The results obtained after removing unnecessary dependencies from the project file by following the directives given by the program we developed for Project-2 are given below, respectively.

As seen in Figure 30, the used program (code) size is 45942 bytes. There was a decrease in code size by 1207 bytes.

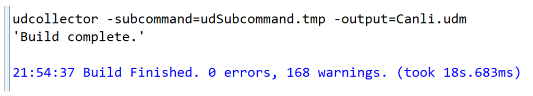
In Figure 31 and Figure 32, the allocation rate of the RAM and ROM areas of the microcontroller in the linker stage at the end of the compilation process is shown. Here, it was observed that the RAM usage decreased from 88.64% to 88.60%. ROM usage decreased from 92.31% to 90.46%. Since functions are stored in ROM and variables are stored in RAM, it is natural for the ROM section to be freed up more than RAM.

As seen in Figure 33, the total build/compile time was measured as 19 seconds and 353 milliseconds. It has been observed that compilation time has decreased significantly as a result of both removing unused but defined functions and removing unnecessary header file dependencies.



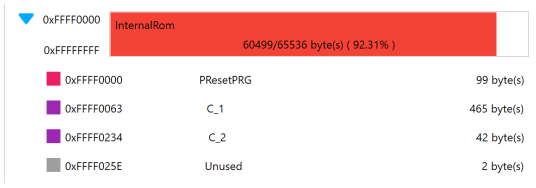
```
0xFFFF0000   InternalRom
0xFFFFFFFF   43786/65536 byte(s) ( 66.81% )
0xFFFF0000   PResetPRG   99 byte(s)
0xFFFF0063   Unused      3997 byte(s)
0xFFFF1000   C_1         447 byte(s)
```

Figure 24: Internal ROM size of the Project-1 after program directives applied



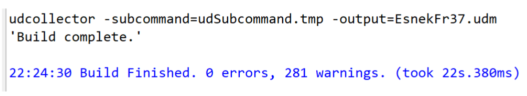
```
udcollector -subcommand=udSubcommand.tmp -output=Canli.udm
'Build complete.'

21:54:37 Build Finished. 0 errors, 168 warnings. (took 18s.683ms)
```

Figure 25: Build time of the Project-1 after program directives applied



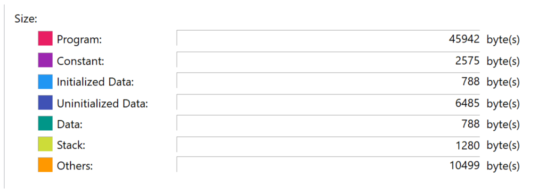
```
0xFFFF0000   InternalRom
0xFFFFFFFF   60499/65536 byte(s) ( 92.31% )
0xFFFF0000   PResetPRG   99 byte(s)
0xFFFF0063   C_1         465 byte(s)
0xFFFF0234   C_2         42 byte(s)
0xFFFF025E   Unused      2 byte(s)
```

Figure 28: Internal ROM size of the Project-2 before program directives applied



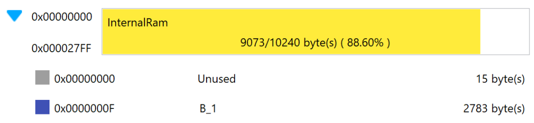
```
udcollector -subcommand=udSubcommand.tmp -output=EsnekFr37.udm
'Build complete.'

22:24:30 Build Finished. 0 errors, 281 warnings. (took 22s.380ms)
```

Figure 29: Build time of the Project-2 before program directives applied



```
Size:
  Program:                45942  byte(s)
  Constant:                2575  byte(s)
  Initialized Data:         788  byte(s)
  Uninitialized Data:      6485  byte(s)
  Data:                     788  byte(s)
  Stack:                   1280  byte(s)
  Others:                 10499  byte(s)
```

Figure 30: Memory size of the Project-2 after program directives applied



```
0x00000000   InternalRam
0x000027FF   9073/10240 byte(s) ( 88.60% )
   0x00000000        Unused                15 byte(s)
   0x0000000F        B_1                 2783 byte(s)
```

Figure 31: Internal RAM size of the Project-2 after program directives applied

C. Project-3

The sample software used in this section belongs to an Oven embedded system project developed in the R&D department of a white goods company. In this project, the STM8L151C6 processor from the STM8 family produced by the STMicroelectronics company, is used. All the results for this project were obtained from the ST Visual Develop IDE.

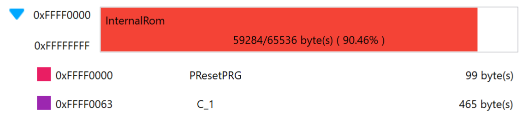
ST Visual Develop IDE does not provide user interface unlike E2 Studio for memory and resource usages as a result of compiling the project file for the specified microcontroller.

Instead, it produces output files such as <project_name>.map, <project_name>.lkf, <project_name>.elf, etc. that contain information about the resource usage, memory allocation, code segments of the project. It can also give only the stack usage of the specified function, as shown in Figure 34, and the address range of the function in memory in detail, as shown in Figure 35. The results obtained without following the directives given by the program we developed for Project-3 will be given in order.

As seen in Figure 36, it has been observed that the used program size is 25384 bytes. The results that are obtained after removing unnecessary dependencies from the project file by following the directives given by the program we developed for Project-3 are given below, in order.

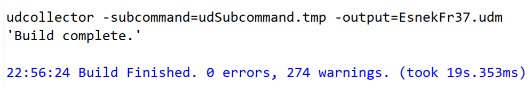
The results obtained after removing unnecessary dependencies from the project file by following the directives given by the program we developed for Project-3 are given below, respectively.

As seen in Figure 37, the used program (code) size is 24657 bytes. There was a decrease in code size by 727 bytes. The compile times data for this and the next project are shown in the "Compile Times" table at Project Compile Times.



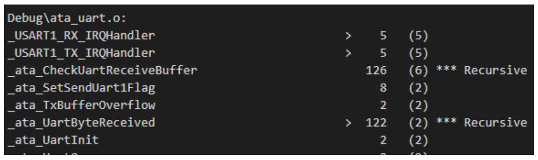
```
0xFFFF0000   InternalRam
0xFFFFFFFF   59284/65536 byte(s) ( 90.46% )
   0xFFFF0000        PResetPRG             99 byte(s)
   0xFFFF0063        C_1                  465 byte(s)
```

Figure 32: Internal ROM size of the Project-2 after program directives applied



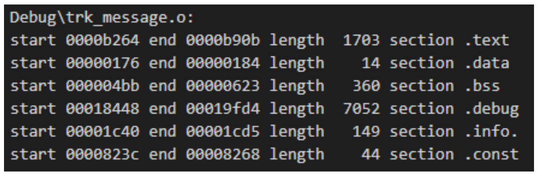
```
udcollector -subcommand=udSubcommand.tmp -output=EsnekFr37.udm
'Build complete.'

22:56:24 Build Finished. 0 errors, 274 warnings. (took 19s.353ms)
```

Figure 33: Build time of the Project-2 after program directives applied



```
Debug\ata_uart.o:
_USART1_RX_IRQHandler                  >    5   (5)
_USART1_TX_IRQHandler                  >    5   (5)
_ata_CheckUartReceiveBuffer               126   (6) *** Recursive
_ata_SetSendUart1Flag                       8   (2)
_ata_TxBufferOverflow                       2   (2)
_ata_UartByteReceived                  >  122   (2) *** Recursive
_ata_UartInit                               2   (2)
```

Figure 34: Example of function Stack usage from Project-3

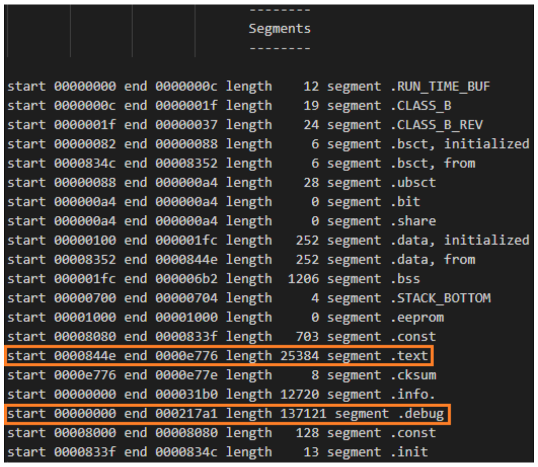


```
Debug\trk_message.o:
start 0000b264 end 0000b90b length  1703 section .text
start 00000176 end 00000184 length    14 section .data
start 000004bb end 00000623 length   360 section .bss
start 00018448 end 00019fd4 length  7052 section .debug
start 00001c40 end 00001cd5 length   149 section .info.
start 0000823c end 00008268 length    44 section .const
```

Figure 35: Example of function Memory address segmentation from Project-3

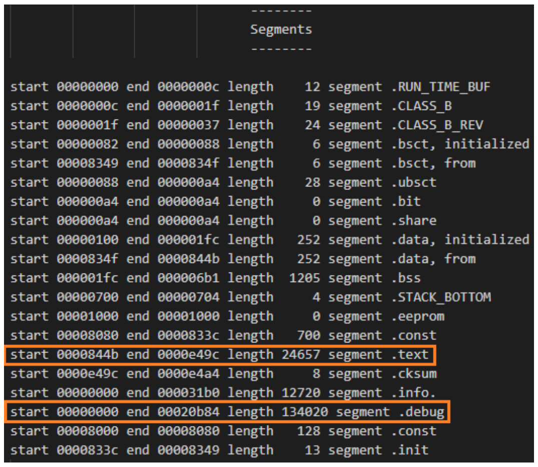
D. Project-4

The sample software used in this section belongs to an Air Conditioner embedded system project developed in the R&D department of a white goods company. In this project, the RL78G13 processor from the RL78 family produced by the Renesas



```
                            --------
                            Segments
                            --------

start 00000000 end 0000000c length    12 segment .RUN_TIME_BUF
start 0000000c end 0000001f length    19 segment .CLASS_B
start 0000001f end 00000037 length    24 segment .CLASS_B_REV
start 00000082 end 00000088 length     6 segment .bsct, initialized
start 0000834c end 00008352 length     6 segment .bsct, from
start 00000088 end 000000a4 length    28 segment .ubsct
start 000000a4 end 000000a4 length     0 segment .bit
start 000000a4 end 000000a4 length     0 segment .share
start 00000100 end 000001fc length   252 segment .data, initialized
start 00008352 end 0000844e length   252 segment .data, from
start 000001fc end 000006b2 length  1206 segment .bss
start 00000700 end 00000704 length     4 segment .STACK_BOTTOM
start 00001000 end 00001000 length     0 segment .eeprom
start 00008080 end 0000833f length   703 segment .const
start 0000844e end 0000e776 length 25384 segment .text
start 0000e776 end 0000e77e length     8 segment .cksum
start 00000000 end 000031b0 length 12720 segment .info.
start 00000000 end 000217a1 length 137121 segment .debug
start 00008000 end 00008080 length   128 segment .const
start 0000833f end 0000834c length    13 segment .init
```

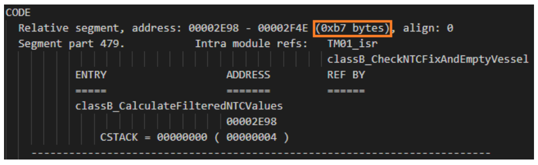
Figure 36: Memory segmentation of the Project-3 before program directives applied



```
                        --------
                        Segments
                        --------

start 00000000 end 0000000c length    12 segment .RUN_TIME_BUF
start 0000000c end 0000001f length    19 segment .CLASS_B
start 0000001f end 00000037 length    24 segment .CLASS_B_REV
start 00000082 end 00000088 length     6 segment .bsct, initialized
start 00008349 end 0000834f length     6 segment .bsct, from
start 00000088 end 000000a4 length    28 segment .ubsct
start 000000a4 end 000000a4 length     0 segment .bit
start 000000a4 end 000000a4 length     0 segment .share
start 00000100 end 000001fc length   252 segment .data, initialized
start 0000834f end 0000844b length   252 segment .data, from
start 000001fc end 000006b1 length  1205 segment .bss
start 00000700 end 00000704 length     4 segment .STACK_BOTTOM
start 00001000 end 00001000 length     0 segment .eeprom
start 00008080 end 0000833c length   700 segment .const
start 0000844b end 0000e49c length 24657 segment .text
start 0000e49c end 0000e4a4 length     8 segment .cksum
start 00000000 end 000031b0 length 12720 segment .info.
start 00000000 end 00020b84 length 134020 segment .debug
start 00008000 end 00008080 length   128 segment .const
start 0000833c end 00008349 length    13 segment .init
```

Figure 37: Memory segmentation of the Project-3 after program directives applied

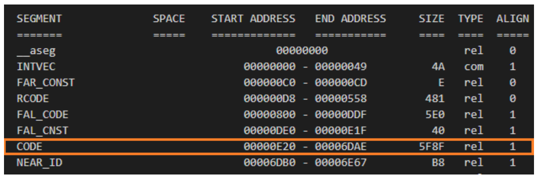
company is used. All the results for this project have been obtained from the IAR Embedded Workbench IDE. IAR Embedded Workbench IDE does not provide user interface like ST Visual Develop IDE for memory and resource usages as a result of compiling the project file for the specified microcontroller. Instead, it produces an output file named <project_name>.map containing information about the resource and memory usage of the project. It can also give a detailed output only about the size of the specified function and the address where the function is located in memory, as shown in Figure 38.



```
CODE
  Relative segment, address: 00002E98 - 00002F4E (0xb7 bytes), align: 0
  Segment part 479.           Intra module refs:   TM01_isr
                                                   classB_CheckNTCFixAndEmptyVessel
          ENTRY                     ADDRESS        REF BY
          =====                     =======        ======
          classB_CalculateFilteredNTCValues
                                    00002E98
            CSTACK = 00000000 ( 00000004 )
    -------------------------------------------------------------------------
```

Figure 38: Example of function memory segment from Project-4

In addition to this, it produces an output that gives all segments in the project in address order. This output is shown in Figure 39.

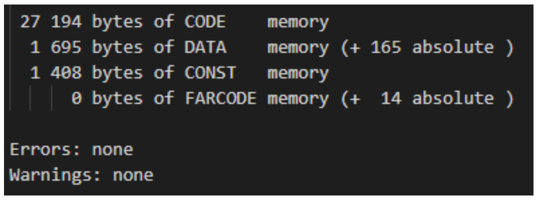


```
SEGMENT              SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======              =====    =============   ===========     ====  ====  =====
__aseg                              00000000                        rel    0
INTVEC                        00000000 - 00000049           4A  com    1
FAR_CONST                     000000C0 - 000000CD            E  rel    0
RCODE                         000000D8 - 00000558          481  rel    0
FAL_CODE                      00000800 - 00000DDF          5E0  rel    1
FAL_CNST                      00000DE0 - 00000E1F           40  rel    1
CODE                          00000E20 - 00006DAE         5F8F  rel    1
NEAR_ID                       00006DB0 - 00006E67           B8  rel    1
```

Figure 39: Segments in address order from Project-4

The results obtained without following the directives given by the program we developed for Project-4 are shown below.

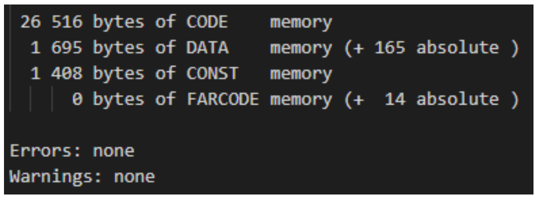
We observed that the used program size was 27194 bytes as shown in Figure 40.



```
 27 194 bytes of CODE     memory
  1 695 bytes of DATA     memory (+ 165 absolute )
  1 408 bytes of CONST    memory
      0 bytes of FARCODE memory (+  14 absolute )

Errors: none
Warnings: none
```

Figure 40: Memory map of the Project-4 before program directives applied

The results obtained for Project-4 after removing unnecessary dependencies from the project file by following the directives given by the program that we developed are shown below.



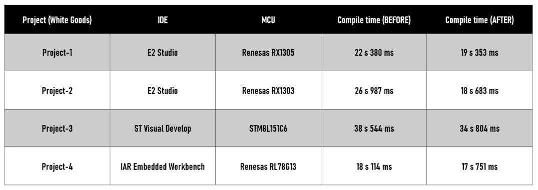
```
 26 516 bytes of CODE     memory
  1 695 bytes of DATA     memory (+ 165 absolute )
  1 408 bytes of CONST    memory
      0 bytes of FARCODE memory (+  14 absolute )

Errors: none
Warnings: none
```

Figure 41: Memory map of the Project-4 after program directives applied

We observed that the used program size was 26516 bytes as shown in Figure 41. There was a 678 byte reduction in code size.

E. Project Compile Times

In this section, we executed the program we developed for 4 different white goods embedded system software projects. We removed all unnecessary dependencies that given by the program as output and put the changes in the compilation times of the projects in a table. With the measurements we have taken, we have proven that the projects are compiled faster after the unnecessary dependencies are removed. Compilation times are given for all projects in the Figure 42.



Project (White Goods)	IDE	MCU	Compile time (BEFORE)	Compile time (AFTER)
Project-1	E2 Studio	Renesas RX1305	22 s 380 ms	19 s 353 ms
Project-2	E2 Studio	Renesas RX1303	26 s 987 ms	18 s 683 ms
Project-3	ST Visual Develop	STM8L151C6	38 s 544 ms	34 s 804 ms
Project-4	IAR Embedded Workbench	Renesas RL78G13	18 s 114 ms	17 s 751 ms

Figure 42: Compile Time Comparison for all projects

V. Related Work

In this section, we will examine other studies that are closely related to our project. In the ”ABC: Accelerated Building of C/C++ Projects” [3] study, researchers who found that unnecessary dependencies lead to unnecessary recompilation in incremental builds, thus slowing down the compilation process significantly, aimed to avoid this problem with a technique that they call bypass compilation. They developed a prototype tool called ABC that detects unnecessary dependencies and unnecessary changes to the compiler and eliminates unnecessary recompiles. The goal here was to speed up the incremental build. With the tool they developed and the approach they adopted, researchers who experimented in some sample projects greatly accelerated the incremental compilation process. Similar aspects of this study with our work are to avoid unnecessary recompilation of compilers due to unnecessary dependencies created by unnecessary added header files to improve compilation time.

In the article ”Optimizing Header File Include Directives” [4], researchers aim to develop an efficient general algorithm for optimizing include file directives. With a commercial tool called Klockwork, they identify violations such as how many times a header file can be added, identify files that cannot be self-compiled, and extra inclusions. Their approach is to process the files compilation units according to the established descriptive association rules and post-process the data to divide the included files into those that are necessary to compile a particular unit and those that are not.

Contrary to the above articles we have reviewed, we do a research on the source code rather than a development on the compiler side by warning the user about the unnecessary dependencies of a project written in C language.

VI. Conclusion

In this project, we developed a program that runs on Linux operating system and finds unnecessary dependencies in projects written in C language. We handled the project in two big steps. The first was to detect all unnecessary header files included to the C source files in the project. The second was to detect all functions declared globally but not used anywhere in the project. We developed our program with Python language. We used ReGex to parse strings in the analyzed files. We found all the function dependencies in the project with the GNU cflow tool. GNU cflow gave us all the function dependencies for the specified project as output in different formats. We found all the functions used in the project from the text format outputs, again using ReGex. By performing both steps in our project, we warned the user about the detected unnecessary dependencies in the project that is given to our program. Finally, we ran our program for 4 separate embedded system projects. We removed unnecessary dependencies according to the outputs we got for each. We observed both a reduction in compile time and a reduction in the size of memory reserved for code as a result of removing unnecessary dependencies from projects.

References

[1] En.wikipedia.org. 2022. Regular expression-Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Regular_expression> [Accessed 30 May 2022].

[2] Medium. 2022. Regex tutorial—A quick cheatsheet by examples. [online] Available at: <https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285> [Accessed 30 May 2022].

[3] Spinellis, D., 2009. Optimizing header file include directives. Journal of Software Maintenance and Evolution: Research and Practice, 21(4), pp.233-251.

[4] Zhang, Y., Jiang, Y., Xu, C., Ma, X. and Yu, P., 2015. ABC: Accelerated Building of C/C++ Projects. 2015 Asia-Pacific Software Engineering Conference (APSEC).

[5] En.wikipedia.org. 2022. GNU cflow-Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/GNU_cflow> [Accessed 30 May 2022].

[6] Poznyakoff, S., 2003. GNU Radius reference manual. Boston, MA: GNU Press.