AB-62

APPLICATION BRIEF

Compiled Code Optimizations for Flash Memories

KEN MC KEE
TECHNICAL MARKETING
ENGINEER

November 1995

Order Number: 292165-002

Information in this document is provided solely to enable use of Intel products. Intel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Intel products except as provided in Intel's Terms and Conditions of Sale for such products.

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel retains the right to make changes to these specifications at any time, without notice.

Contact your local Intel sales office or your distributor to obtain the latest specifications before placing your product order.

MDS is an ordering code only and is not used as a product name or trademark of Intel Corporation.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

*Other brands and names are the property of their respective owners.

Additional copies of this document or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

or call 1-800-879-4683

© INTEL CORPORATION 1995

CG-041493

1.0 INTRODUCTION

Intel's high-performance flash memories, the 28F016XS and 28F016XD, are changing the typical view of nonvolatile memories and moving into applications previously dominated by fast volatile memories such as DRAM and SRAM. These high-performance flash components were specifically designed with optimized system interfaces to deliver high read performance to embedded applications. Systems that in the past shadowed code from slow nonvolatile memory into fast volatile memory to improve performance can now eliminate this memory redundancy by utilizing flash memory. The CPU can now execute code directly out of a nonvolatile memory without slowing down the overall system. Code stored in high-performance flash memory will in many cases execute faster than that in DRAM. This new approach reduces system component count, cost and power consumption and improves overall system read performance and reliability.

One important aspect to keep in mind when developing code for direct execution out of a nonvolatile memory concerns code and data segments—they must be separated! This separation prevents program data from accidentally being written to nonvolatile memory. Flash memory is in-system updateable, but not fully bit alterable. Therefore, specific address assignments are necessary to fit code and data into a given target system's memory map.

The rest of this application brief will describe the process of developing optimized code for direct execution out of flash memory, as well as recompiling legacy code. For specific component information about high-performance flash memories, reference the Additional Information section.

2.0 EMBEDDED ENVIRONMENT

Figure 1 illustrates a typical embedded system, consisting of a CPU, random access memory (RAM), some type of mass storage device and memory mapped input/output (I/O). In this model, RAM contains both code and data segments and the nonvolatile memory stores the program when power is removed from the system. On power-up, an initialization routine copies the stored program resident in the nonvolatile memory into RAM for execution. Figure 2 shows an optimized system utilizing high-performance flash memory for both code storage and execution. This improved system utilizes memory resources more efficiently, thereby reducing the system memory component count and cost.

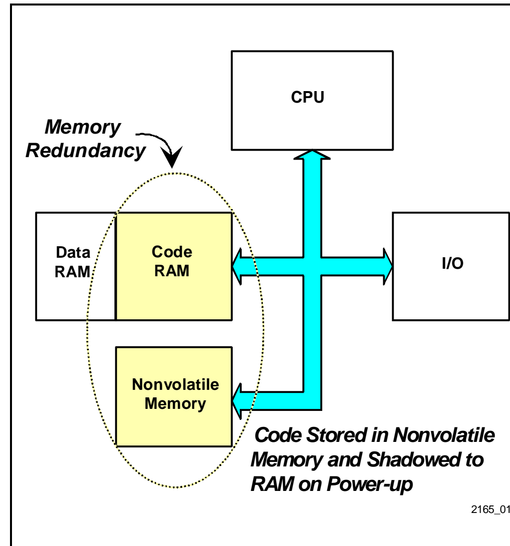

Figure 1. Code Shadowing Causes Memory Redundancy Which Increases the Overall System Component Count and Cost

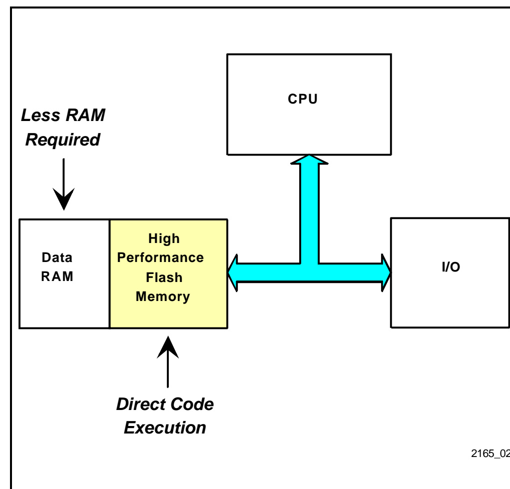

Figure 2. Executing Code Directly out of High-Performance Flash Memory Improves System Performance and Reliability, While Reducing Overall System Cost

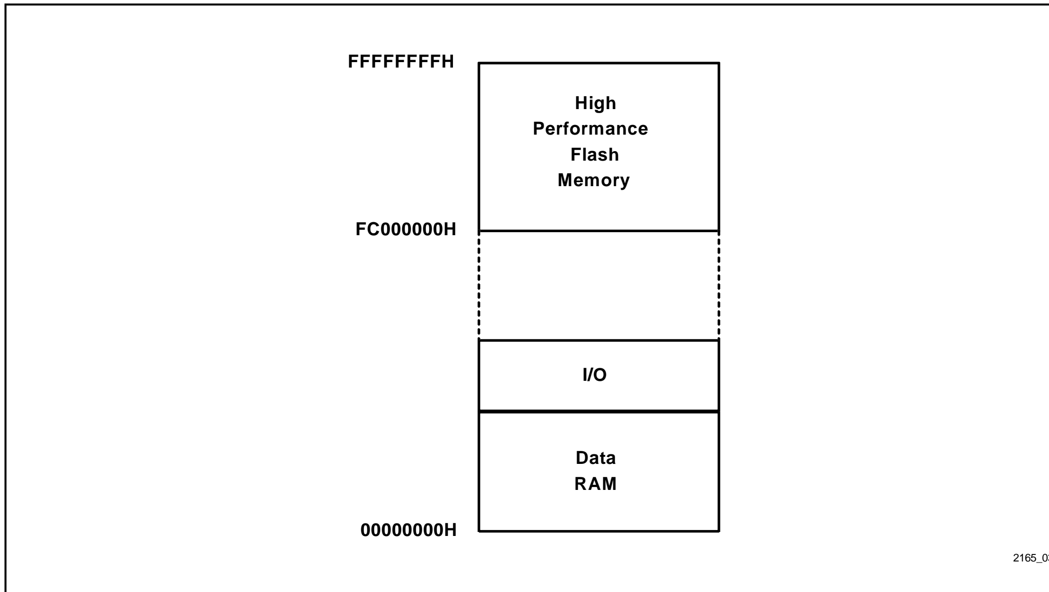

Figure 3. Example Memory Map for Optimized Embedded Environment Illustrated in Figure 2

Figure 3 shows the resultant memory map for the high-performance flash memory design illustrated in Figure 2. Notice that each of the three different elements within this optimized system resides at a unique address. Unlike a PC environment that is well defined in terms of its memory map configuration, the embedded environment varies from one system to another. As a result, code for embedded applications must be specifically developed to fit a particular system architecture.

For example, the embedded system memory map illustrated in Figure 3 executes code directly out of the flash memory. Therefore, the code segment points to flash memory at address FC000000H. The data segment, on the other hand, points to RAM at address 00000000H. This segment will contain program data such as the stack and program variables that are frequently updated.

3.0 SEGMENTING CODE AND DATA

The compiler is an important tool in developing optimized code for flash memory. It's the role of the compiler to convert source code into the actual machine language for the target microprocessor. This tool, however, only generates machine code. It does not resolve memory allocation addresses for code and data segments. This job is left up to the linker to complete.

Fortunately, many linkers provide a mechanism to place code and data in a specific location explicitly for embedded applications. Therefore, the source code structure does not have to define specific locations for data structures when generating code for direct execution out of a nonvolatile memory. This linker mechanism furnishes a simple process for porting code from one system to another without requiring any code modifications. The following sections will step through the process of segmenting code, utilizing different development tools.

Before continuing, however, it's important to understand that the code segment should be set-up as a "*read only*" segment. This will prevent writing information to the code segment. Remember, flash memory is in-system updateable but is not fully bit alterable. Therefore, self-modifying code should be avoided.

Compilers/Linkers for Embedded Applications

Many compilers/linkers for embedded microprocessors, such as the GNU/960 tool set for the i960® microprocessor family, specify code and data segments locations via a command line switch. Through the command line, the linker receives specific segment addresses. The linker then uses this information to place the code and data segments into the target system.

The GNU/960 linker, for example, incorporates a -Ttarget command line option that invokes the linker to search for a file entitled *TARGET.LD*. This file defines the target system's memory map, informing the linker where to place the program's code and data segments in the target system. Figure 4 illustrates a *TARGET.LD* file for the embedded environment shown in Figures 2 and 3.

```
MEMORY
{       Flash:      o=0xFC000000,1=0x4000000
        DataRAM:    o=0x00000000,1=0x0010000
}

SECTIONS
{       .text:                     ; Code Segment
        {
        } >Flash

        .data                      ; Data Segment
        {                          ; Initialized Data Variables
        _ram = .;
        } >DataRAM

        .bss:                      ; Data Segment
        {                          ; Un-initialized Data
        } > DataRAM                ; Variables
}
```

2165_04

Figure 4. Example *TARGET.LD* File for the Embedded Environment in Figures 2 and 3

With the code and data segments defined, the initialized data stored in the nonvolatile memory that the program will update when executing must be linked to the data segment upon invoking the program. To handle this procedure, a small routine must be added to the code. Fortunately, a myriad of compilers/linkers for embedded applications provide a one-step process to setting up initialization tables. The linker integrates a vendor-developed start-up module which upon power-up copies all initialized data to the appropriate RAM locations for system execution. This linker utility eliminates additional development time in creating optimized code for execution out of flash memory.

Data structures that require no modification during the run time of the program, such as static data tables and constant variables, can remain situated within the nonvolatile memory. The program will only read these types of constant data structures. Leaving these structures within the nonvolatile memory provides a positive system benefit in that it reduces the necessary amount of available RAM required by the program.

Compilers/Linkers for PC Platforms

When developing code for Intel Architecture microprocessors, standard DOS compilers may be used to develop optimized code for direct execution out of flash memory. High level compilers such as C and C++, with the help of two utilities called "linking locator" and "start-up," can generate code suitable for the embedded world.

The job of the linking locator is to place code and data segments into the target memory map, given specific input parameters. The passing of parameters is accomplished in the fashion as explained in the previous section "Compilers/Linkers for Embedded Applications." Next, the "start-up" utility enables the embedded code to be initiated without DOS being present in the embedded system. The "start-up" routine explicitly assigns the STACK segment as an independent segment so that it will not point to nonvolatile memory. Figure 5 illustrates an example of a "start-up" routine for Microsoft C programs called *INVOKEC.ASM*. The steps involved in creating optimized code using these tools is very minimal.

Following the flowchart in Figure 6, the initial development phase requires no alteration to standard practices involved in developing code for a PC platform. The major advantage here is that the development and program debugging can be accomplished independently on a PC before moving the code into an embedded system.

Functional at the PC platform level, the code is once again compiled and linked. During the linking process, the object file, embedded libraries and "start-up" utilities are tied together with the "linking locator" to segment code and data to fit the target memory map.

The embedded libraries supplement standard DOS libraries that call on the functionality of the operating system. In the embedded environment, the operating system may not be present. These functions include simple library calls such as *putchar()*. The code for these embedded libraries can be purchased from companies

such as Phar Lap Inc. and Systems & Software Inc. (SSI) or created for the target system.

```
INIT_TEXT SEGMENT PUBLIC DWORD 'CODE'
INIT_TEXT ENDS

STACK  SEGMENT STACK DWORD 'STACK'
SSTACK  DB     1024 DUP (?)          ; Start (bottom) of stack.
ESTACK  LABEL  BYTE                  ; End (top) of stack.
STACK  ENDS

_DATA SEGMENT PUBLIC DWORD 'DATA'
ARGV  DW    OFFSET PNAME
PNAME  DB   'main',0
_DATA  ENDS

DGROUP  GROUP _DATA

   public __acrtused
__acrtused   equ    0

   EXTRN  _main:FAR

   ASSUME CS:INIT_TEXT,DS:DGROUP,SS:STACK

INIT_TEXT SEGMENT

   PUBLIC start,_exit

_exit  PROC   FAR
    jmp    $                         ; Hang here upon exit.
_exit  ENDP

start  PROC  FAR

    mov    ax,SEG DGROUP             ; Initialize DS
    mov    ds,ax                     ; and ES to point
    mov    es,ax                     ; to the data segment.
    mov    ax,SEG STACK              ; Initialize Stack Segment
    mov    ss,ax                     ; register and set up the
                                     ; stack.
    mov    sp,OFFSET STACK:ESTACK
    push   ds                        ; Set up
    mov    ax,OFFSET D GROUP:ARGV   ; a  C standard
    push   ax                        ; environment
    mov    ax,1                      ; for calling
    push   ax                        ; the C program.
    call     _main                   ; Invoke the program.
    call     _exit                   ; go to the exit routine
             ; when done
start  ENDP
INIT_TEXT ENDS
END      start
```

2165_05

Figure 5. Example "Start-Up" Routine Enabling Optimized Code for Flash Memory to Initiate without DOS Being Present

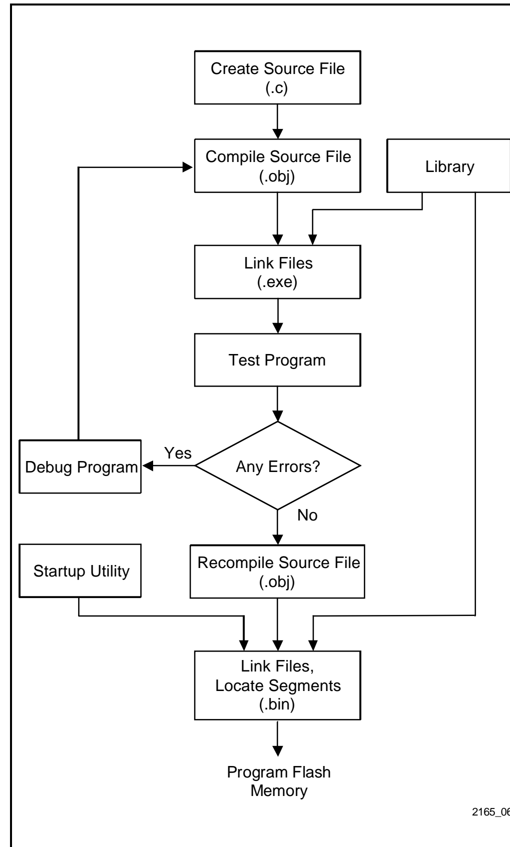

Figure 6. Flowchart for Developing Optimized Code for Flash Memory Using Off-the-Shelf C Compilers for Intel Architecture Based Designs

4.0 OPTIMIZING LEGACY CODE

In most situations, it may be desirable to utilize existing code from a previous design. This practice can drastically reduce a new design's development cycle. But, can this legacy code port into a new system environment that utilizes the benefits of high-performance flash memories? The answer is emphatically YES!

Depending upon the system's architectural changes and legacy code structure, different steps may or may not be necessary to optimize the legacy code for flash memory usage. The following sections will describe the aspects requiring consideration when utilizing legacy code.

Legacy Code That Executed out of Nonvolatile Memory

Many systems today already execute code directly out of nonvolatile memory. When these types of systems advance to employ the benefits of high-performance flash memory, the legacy code may be put to use with no alterations or recompiling necessary.

Legacy Code That Executed out of RAM

In high-performance embedded systems that shadow code into RAM, the code and data segments may not be separated in an appropriate fashion for direct execution out of high-performance flash memory. When shadowing code into RAM, the data segment can immediately follow the code segment without posing a system problem because the code segment resides in the RAM area. In fact, linkers typically default to this type of configuration if not informed otherwise. Figures 7 and 8 illustrate two different system implementations of shadowing code for a nonvolatile memory.

In an optimized environment, the data segment cannot point to a nonvolatile memory. To overcome this hurdle, two specific areas need examination when converting this type of legacy code. First, the initial routine that copies the code stored in nonvolatile memory into RAM on power-up will no longer be needed. The code will now execute directly out of the storage memory. This elimination will reduce code size and unnecessary software overhead. Second, explicit address assignments, if implemented in the legacy code, will require examination. The address assignments for program elements such as data structures and variables must be configured to fit the new embedded target system. Otherwise, these assignments can be removed and replaced with variable references which the compiler and linker can then use to place in appropriate address locations.

After optimizing the code for flash memory, the legacy code will require recompiling. See Section 3.0 for information about compiling optimal code for flash memory.

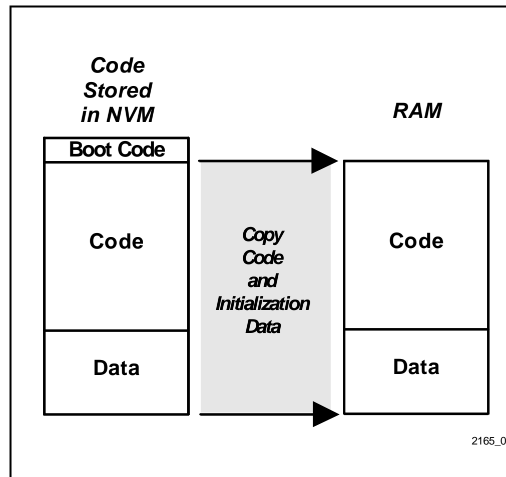

Figure 7. Code Stored in Nonvolatile Memory, Then Copied into RAM for Execution

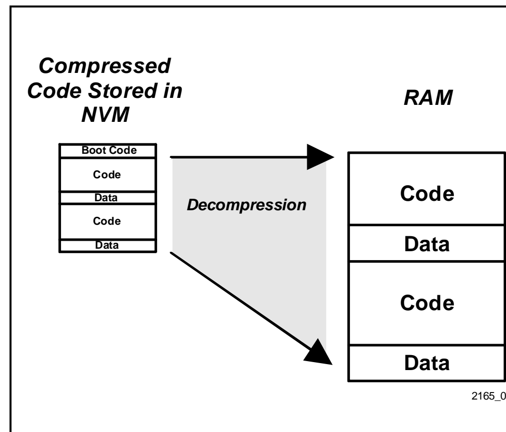

Figure 8. Compressed Code Stored in Nonvolatile Memory, Then Decompressed into RAM for Execution

5.0 DEBUG SUPPORT

Software debugging can consume a great deal of time and effort. However, debugging tools available today can dramatically reduce this time investment. They provide the capability of single stepping, code disassembly, breakpoint insertion and many more features. These features aim to ease the debugging process and are equally applicable for debugging code that executes directly out of either RAM or flash memory.

The only noticeable difference between debugging code out of RAM verses a nonvolatile memory is the breakpoint insertion methodology used by debugging tools. When debugging code in RAM, the instruction at the desired breakpoint reference location is saved. Then, a break instruction is written back to the reference location. This write operation cannot be accomplished in a nonvolatile memory. Consequently, an alternative method must be used by the debugging tool to employ breakpoints when debugging code in nonvolatile memory. The tool set therefore makes use of "*hardware breakpoints,*" thereby utilizing the internal debugging capabilities of the host CPU. Instead of inserting actual breakpoints in the executing code, the desired reference location is stored in the processor's debug registers. For example, the Intel486™ microprocessor provides access to four debug registers. When the processor's instruction pointer reaches the address location stored in the processor debug register, the program will stop execution. At this point, the program's functionality can be examined.

If the host CPU lacks sufficient integrated hardware debugging support, several other strategies can be used to debug execute-in-place (XIP) code stored in flash memory. ROM or flash memory emulators provide one such solution. Memory emulators integrate enhanced debugging features such as built in hardware breakpoint support. The Reference section of this application brief lists example memory emulator vendors.

Another approach is debugging code in RAM. Using this approach, the standard software emulator breakpoint insertion method is valid. The debugged code then ports directly to the optimized XIP embedded system. This approach is most easily achieved in systems that place DRAM on SIMMs, where a temporary increase in the amount of RAM requires no system hardware changes.

A final approach makes use of hardware CPU emulators versus software emulators. CPU emulators provide a great deal of flexibility in breakpoint methodology without writing to the main memory subsystem.

6.0 SUMMARY

This application brief has discussed the process of developing optimized code for high-performance flash memory. The important aspect to remember is that compilers/linkers today provide the capability to produce optimized code for direct execution out of flash memory. For further information about high-performance flash memories, compilers/linkers and linking locators, consult the Additional Information and References sections.

ADDITIONAL INFORMATION

Order Number	Document/Tool
297372	16-Mbit Flash Product Family User's Manual
290532	28F016XS 16-Mbit Synchronous Flash Memory Datasheet
290533	28F016XD 16-Mbit DRAM-Interface Flash Memory Datasheet
292147	AP-348, "Designing with the 28F016XS"
292126	AP-377, "16-Mbit Software Drivers"
292131	AP-384, "Designing with the 28F016XD"
292146	AP-600, "Performance Benefits and Power/Energy Saving of 28F016XS-Based Designs"
292163	AP-610, "Flash Memory In-System Code and Data Update Techniques"
292168	AP-614, "Using the 28F016XD in Embedded PC Designs"
297500	"Interfacing the 28F016XS to the i960® Microprocessor Family"
297504	"Interfacing the 28F016XS to the Intel486™ Microprocessor Family"
292152	AB-58, "28F016XD-Based SIMM Designs"
297508	FLASHBuilder Utility
Contact Intel/Distribution Sales Office	28F016XS and 28F016XD Benchmark Utilities
Contact Intel/Distribution Sales Office	28F016XS and 28F016XD iBIS Models
Contact Intel/Distribution Sales Office	VHDL Models for the 28F016XS and 28F016XD
Contact Intel/Distribution Sales Office	Timing Designer Files for the 28F016XS and 28F016XD
Contact Intel/Distribution Sales Office	28F016XS and 28F016XD Orcad and ViewLogic Schematic Symbols

REVISION HISTORY

Number	Description
001	Original version
002	Changed title from "Compiled Code Optimizations for Embedded Flash RAM Memories" to "Compiled Code Optimizations for Flash Memories" Removed all Embedded Flash RAM memory references

REFERENCES

NOTE:

The compiler/linker listing below is just a sample of available embedded tools that have the capability to segment code and data. Since these embedded tools continuously improve, Intel recommends that designers contact vendors for updated information about compiler/linker capabilities and new user utilities. Intel will continue to work with the industry to develop optimum solutions for software compilation. Intel Corporation assume no responsibility for the quality or reliability of these software products. No patent licenses are implied.

Compilers/Linkers for Embedded Applications

Cygnus
1937 Landing Drive
Mountain View, CA 94043
(617) 872-3296

Embedded Performance, Inc.
1860 Barber Lane
Milpitas, CA 95635
(408) 434-2210

GreenHills
510 Castillo Drive
Santa Barbara, CA 93101
(805) 965-6044

MetaWare
2161 Delaware Avenue
Santa Cruz, CA 95060
(408) 429-6382

Microtec Research
2350 Mission College Boulevard
Santa Clara, CA 95054
(800) 950-5554

Software Development Systems, Inc.
815 Commerce Drive, Suite 250
Oak Brook, IL 60521
(800) 448-7733

Linking Locators and Embedded Libraries for Off-the-Shelf PC Compilers

Phar Lap Inc.
60 Aberdeen Avenue
Cambridge, MA 02138
(617) 661-1510

Systems & Software Inc. (SSI)
18012 Cowan, Suite 100
Irvine, CA 92714
(714) 833-1700

Memory Emulators

Grammar Engine, Inc.
921 Eastwind Drive Suite 122
Westerville, OH 43081
(800) 776-6423

J&M Microtek, Inc.
83 Seaman Road
W Orange, NJ 07052
(201) 325-1892

Tech Tools
P.O. Box 462101
Garland, TX 75046
(214) 272-9392

Filename:	292165_2.DOC
Directory:	C:\TESTDOCS\DOCS
Template:	C:\WIN\WINWORD6\TEMPLATE\ZAN____1.DOT
Title: E	
Subject:	
Author:	Ken Mckee
Keywords:	
Comments:	
Creation Date:	09/20/95 5:17 PM
Revision Number:	16
Last Saved On:	11/28/95 9:49 AM
Last Saved By:	Ward McQueen
Total Editing Time:	189 Minutes
Last Printed On:	11/28/95 9:51 AM
As of Last Complete Printing	
Number of Pages:	11
Number of Words:	2,914 (approx.)
Number of Characters:	16,615 (approx.)