ASM STARTUP CODE

Assembly Language Startup Source Code

Version 1.00

USER'S GUIDE

MoonWare Shareware
16005 Pointer Ridge Drive
Bowie, MD 20716-1744

raymoon@moonware.dgsys.com

LIMITED WARRANTY STRTUP10 is sold AS IS.

MoonWare makes no representations or warranties, expressed or implied, about the contents hereof, including, without limitations, any implied warranties of merchantability or fitness for a particular purpose, all of which MoonWare Shareware expressly disclaims. Neither MoonWare Shareware, nor anyone involved in the distribution of this software shall be liable for any direct, indirect, consequential, or incidental damages arising out of the use, the results of use, or inability to use such product even if I have been advised of the possibility of such damages or claim.

USER'S GUIDE COPYRIGHT NOTICE

Only registered users for their personal use may reproduce this manual by printing it. Non-registered users may not duplicate this USER'S GUIDE in any form.

SOFTWARE COPYRIGHT NOTICE

The ASM Startup assembly code is copyright © 1997 by Raymond Moon. Non-registered users may use this software for a 30-day trial period. At the end of that 30-day trial period, you must become a registered user or stop using the ASM Startup assembly code.

SHAREWARE DISTRIBUTION

ShareWare is not FreeWare. I have chosen this method of distributing the ASM Startup assembly code so you, the user, can use it for a trial period to see if it meets your needs. If it does, you must become a registered user. Registered users will receive the latest version if they registered an older version, or they will receive a free upgrade to the next version. Registered and non registered users are authorized to upload the zip file, STRTUP10.ZIP, to any Bulletin Board System as long as the zip file has not been modified. Please see User Responsibilities, page 4, for more details.

CHARGES FOR DISTRIBUTION

STRTUP10.ZIP cannot be sold without the author's prior written approval. If a third party distributes STRTUP10.ZIP on a diskette or CDROM, which third party can charge a small fee not to exceed $5.00 per disk or $25.00 for CDROM to cover the cost of the media and any shipping costs.

TRADEMARK INFORMATION

Microsoft is a registered trademark of Microsoft Corporation.

PKWARE, PKZIP and PKUNZIP are registered trademarks of PKWARE, Inc.

TABLE OF CONTENTS

Figures

INTRODUCTION

1.0 What is STRTP10

Back in 1986, I started to write stand alone 100% assembly language programs. Having been programming in C, I wanted to start assembly language programs as I started C programs, that is with the MAIN procedure. Unfortunately, assemblers do not come out of package with startup code, so I set out to develop that code. The goal was to have startup code that provided a uniform environment in which to start assembly language programs. This uniform environment would include: (1) the command argument(s) in the form of `argc` and `*argv[]`; (2) the environmental strings in the form of `*envp[]`; (3) global variables that are initialized to system and program global information; and (4) unneeded memory released. All memory models, all processors and executable formats would be supported. Lastly, the code would reside in .lib files so all the work would be done by the linker.

1.1 Summary of Features

The assembly language startup code performs the following functions:

- Initializes the following global variables:
  - `DGRP`, segment address of DGROUP;
  - `STACK_BOTTOM`, offset to stack bottom in DGROUP;
  - `PSP`, segment address of the PSP;
  - `NEXTPARA`, segment address of the next available memory paragraph;
  - `ENVIRON`, segment address of the program's ENVIRONMENT;
  - `OSMAJOR`, the integer part of the OS system;
  - `OSMINOR`, the decimal part of the OS system; and
  - `ENV_STR_LEN`, length of environmental strings.
- Initializes DS and ES segment registers to DGROUP;
- Shrinks memory down to the size of the program by releasing all memory above the program;
- Supports returning an exit code to the operating system;
- Provides the count of command line arguments in the form of `argc`;
- Provides the executable's drive, full path, filename and extension as `*argv[0]`;
- Provides pointers as `*argv[]` to the command line arguments parsed in ASCIIZ format;
- Provides pointers as `*envp[]` to the environmental variables parsed in ASCIIZ format;
- Supports both .exe and .com executable formats; and
- Supports all memory models from tiny to huge.

Since not all of these features are needed in every program I write, four versions of the startup code were written. They are:

- Version 0, STARTUP0.ASM, supports all above except `argc`, `*argv[]` and `*envp[]` features;
- Version 1, STARTUP1.ASM, supports all of Version 0 plus `argc` and `*argv[]` features.
  - `*argv[0]` is the first command line argument vice the program name;
- Version 2, STARTUP2.ASM, supports all of Version 1, and `*argv[0]` is the program name; and
- Version 3, STARTUP3.ASM, supports all the above features.

There are two support files that contain all the global data initialized by the startup code. These files are:

SUDATA.ASM declares all global variables common to all startup procedures. These global variables are:

DGRP, segment address of DGROUP;
STACK_BOTTOM, offset to stack bottom in DGROUP;
PSP, segment address of the PSP;
NEXTPARA, segment address of the next available memory paragraph;
ENVIRON, segment address of the program's ENVIRONMENT;
OSMAJOR, the integer part of the OS system; and
OSMINOR, the decimal part of the OS system.

SUDATA3.ASM declares the global variable unique to STARTUP3.ASM that is:

ENV_STR_LEN, length of environmental strings.

Included are templates and include files to facilitate the use of the startup code:

MAIN.ASM, an assembly language template for the MAIN procedure of a program using any version of the startup code;
ASMPROC.ASM, an assembly language template for non MAIN procedures;
LIB.INC, an include file that defines the correct ALIB? Library and all library procedures according the memory model;
PROCESOR.INC, an include file that specifies the target processor for which the source codes will be assembled; and
STARTUP.INC, an include file specifies references to all global variables specified by the startup code.

So that you can see how to write 100% assembly language programs, I have included the following source code files. These source code files were used to write demonstration programs that illustrate all the features of the startup code. The last four procedures are general-purpose procedures that can be used in any program. These files are:

DEMO0.ASM, MAIN procedure that demonstrates all the features of version 0 of the startup code;
DEMO12.ASM, MAIN procedure that demonstrates all the features of versions 1 and 2 of the startup code;
DEMO3.ASM, MAIN procedure that demonstrates all the features of version 3 of the startup code;
HEX2BIN.ASM, a procedure that converts a binary word to an ASCIIZ hexadecimal string;
PRINT.ASM, a procedure that displays the passed string on the console;
PUTS.ASM, a procedure that displays the passed string on the console and then sends a carriage return and line feed so the cursor will be at the start of the next line; and
UTOA.ASM, a procedure that converts an unsigned word to an ASCIIZ decimal string.

Lastly, to make the use of this code even easier, I have included .lib files with the above procedures included except as noted. The library files are:

ALIBT.LIB, a library file of all the above source files, except MAIN and startup procedures, assembled in the tiny memory model;
ALIBS.LIB, a library file of all the above source files, except MAIN procedures, assembled in the small memory model;
ALIBC.LIB, a library file of all the above source files, except MAIN procedures, assembled in the compact memory model;
ALIBM.LIB, a library file of all the above source files, except MAIN procedures, assembled in the medium memory model;

ALIBL.LIB, a library file of all the above source files, except MAIN procedures, assembled in the large memory model;

ALIBH.LIB, a library file of all the above source files, except MAIN procedures, assembled in the huge memory model; and

1.2 Cost of Using This Startup Code

The cost of using Startup code depends upon the memory model and `argv[]` addressing chosen. The size of the code and data in bytes are:

Version	Code	Data	Total
0	74- 84	12	86- 96
1	162-195	12	174-207
2	218-247	12	230-259
3	365-405	74	439-479

1.3 File Locations

The Startup files are located in the following directories if the \d switch is used to unzip STARTUP.ZIP, the .zip file within STRTUP10.ZIP.

```
README.TXT
    \BATS
        MAKELIBS.BAT
        RMLIBS.BAT
        SU.BAT
        SUS.BAT
    \DEMO
        DEMO0.ASM
        DEMO12.ASM
        DEMO3.ASM
    \DOC
        STARTUP.PDF
    \INCLUDES
        LIB.INC
        PROCESOR.INC
        STARTUP.INC
    \LIBS
        ALIBT.LIB
        ALIBS.LIB
        ALIBC.LIB
        ALIBM.LIB
        ALIBL.LIB
        ALIBH.LIB
    \LIBSRC
        BIN2HEX.ASM
        PRINT.ASM
        PUTS.ASM
        STACK.ASM
        STCKCHK.ASM
        UTOA.ASM
    \STARTUP
        STARTUP0.ASM
        STARTUP1.ASM
        STARTUP2.ASM
        STARTUP3.ASM
        SUDATA.ASM
        SUDATA3.ASM
    \TMPLATES
        ASMPROC.ASM
        MAIN.ASM
```

1.4 Assembler Requirements

The current version of the startup code was developed on Microsoft MASM version 6.11d. Suitability of the code for any earlier version of MASM or other assemblers has not been verified.

1.5 Target Program Requirements

The programs produced using this version of ASM Startup Code are required to execute on versions of DOS 3.0 or later.

1.6 User Responsibilities

As mentioned earlier, ShareWare is **not** FreeWare. You, as the user, have a 30 day trial period to evaluate the ASM STARTUP code and associated supporting files. You are **not** authorized to distribute programs based upon any code included in STRTUP10.ZIP during this trial period. If the code meets your needs and you want to continue to use the code to write programs, you **must** register your copy of the ASM STARTUP code. When you register, there are several options that are explained below.

Personal Registration—The basic Non-Commercial Registration is $10.00 that allows you to use ASM Startup Source Code on one computer at a time for personal use. With a personal license, you may not distribute any of programs, whether commercial, ShareWare or FreeWare, based any code from ASM STARTUP.

Commercial Registration—The basic Commercial Registration is $20.00 that allows use ASM STARTUP Code on up to ten stand alone or networked computers for commercial use. Registration for each additional computer is $1.00 each. With a commercial license, you can distribute any programs using ASM Startup Source Code as long as my copyright appears in the documentation and the source code itself is **not** distributed without prior agreement of the author.

Enterprise Registration—Contact the author.

Any user, registered or non-registered, may upload the original and unmodified STRTUP10.ZIP file to any Bulletin Board System. I use the authentication feature of PKWARE’S ZIP program. If unzipping STRTUP10.ZIP with PKWARE’s UNZIP software does reveal the -AV after each filename, the file has not been modified. STRTUP10.ZIP may be distributed on diskettes or CDROM if no more than $5.00 per disk or $25.00 for CDROM to cover the cost of the media and any shipping costs.

1.7 Authenticity of Startup Code

To ensure yourself the you have an authentic version, check that all files -AV after each file when you unzip STRTUP10.ZIP. If your version does not unzip with an -AV after each file, please contact the me with the location where you got the file. I can be contacted at the addresses below.

1.8 User Comments

User comments are appreciated at any time. If your comments result in a product improvement, you will receive a free upgrade. Please send any comments to:

E-mail
raymoon@moonware.dgsys.com
Mail
Raymond Moon
16005 Pointer Ridge Drive
Bowie, MD 20716-1744

USING ASM STARTUP CODE

2.0 Purpose

The purpose of this chapter is to provide you with enough information to get started, i.e., selecting the correct version, and explaining the templates provided. You will find references to more detailed discussions later in the User's Guide. I did this so that you will not be bogged down in detail, but the detail is available if desired. The documentation is written assuming that the reader is at least an intermediate x86 assembly language programmer.

2.1 Required Decisions to Start Using the ASM Startup Code

To use the Startup Code, you will need to determine your program's requirements. To get started, the following information is needed:

a. Which Version of the Startup code your program requires;

b. Which Executable File Format, .exe or .com, your program will use; and

c. If Version 1, 2 or 3, whether your program will use normal or simplified `*argv[]` addressing.

This next section will provide you with the information so that you can answer these questions.

2.2 Criteria for Selecting Which Version

2.2.1 Version 0

This version is used if your program will not use any command line input or environmental strings. This is the simplest version. All that is done is to shrink memory down to the program size, initialize some global variables and set up the segment registers, DS, ES and SS to point to DGROUP.

This last point is important so that automatic variables, created at runtime on the stack, and static variables declared in DGROUP can be addressed as offsets in the same segment. Therefore, pointers can be referenced to DS, ES or SS segment registers. You do not have to worry about the differences between pointers to automatic or static variables. (See .exe Startup Environment, page 12, for a more detailed discussion.) The point here is that if the DS segment register contains the segment value of DGROUP, you can use any pointer to automatic variable with out using the segment override. This is true for all versions of the ASM Startup Code.

2.2.2 Version 1

If your program needs to process the command line, then you need to use Version 1. This version parses the command line onto the stack and then places the count and pointers to each command line arguments onto the stack. This provides the MAIN procedure with the count and command line arguments similar to `argc` and `*argv[]` in C. The only difference is that `*argv[0]` is the first command line argument vice the program name as `*agrv[0]` is in C.

Also, in C, `*argv[]` is a pointer to an array of pointers. To address an individual string in C is rather easy as the compiler generates the code for the extra level of indirection, i.e., loading a pointer to get another pointer to the desired data. In assembly language, this gets very tedious when you have to code this yourself every time you want to address a command line argument. That is why I developed what I call the simplified `*argv[]` addressing technique. Using this technique, `*argv[]` is the address of the first pointer and all subsequent pointers are addressed as offsets from that base pointer. See `*argv[]` Addressing, page 25, for a more detailed discussion.

2.2.3 Version 2

If your program requires the drive, full path, filename and extension of the executing program, you need to use Version 2, which provides this information as `*argv[0]`.

Now, why would this information be useful to a program? Too often, I have seen environmental variables required for users to specify the programs where the home directory is located so that support files can be located. Well, the home directory has been available since Version 2 of DOS. Just parse `*argv[0]`, to obtain the drive and full path. Then filenames of supporting files can be appended to this drive and path to open those files. DOS provides the drive and full path whether the program was evoked using the full path or not. So now, why ask the user to provide information that DOS provides to your program every time it is evoked.

2.2.4 Version 3

If your program needs ready access to the environmental strings, you need to use Version 3 of the ASM Startup Code. This version parses the environmental strings, places an array of pointers to the strings on the stack and, lastly, provides a pointer to that array of pointers as `*envp[]` to the MAIN procedure.

2.3 Selecting the Executable File Type

Next, you must decide if your program will be in the .com or .exe executable file format. This is entirely up to you. If it is a very small program, use the .com file format. See Possible Problem With Available Memory And .com Files, page 30, for more information. The .exe file format gives more flexibility and, therefore, is recommended.

The only difference from the programming perspective is that the .obj file for the startup code must be the first module seen by the linker. The reason is that the startup code has the program origin and must be the first module in the executable image. If you are using Microsoft's Workbench, it is easiest to include the startup code as the first source file in the build.

2.4 Normal or Simplified `agrv[]` Addressing

Lastly, you must decide whether you are going to use normal or simplified `*argv[]` addressing if you are using versions 1 through 3. I recommend the simplified addressing because it requires less coding. See the discussion on `*argv[]` Addressing on page 25. To select simplified addressing, just define the label, SIMPLE, using the /D MASM command line option or the Assembler Options in Microsoft's Workbench.

2.5 MAIN Procedure Template

Now, that you have selected the version, file format, and addressing, you can start developing your MAIN procedure. I have included a template for that MAIN procedure. To understand the template, the following sections explain each section.

2.5.1 Background

The first thing you will notice is that I believe in documentation. I will not tell you how many times good documentation has helped me in the past 12 years of assembly language programming. Assembly language code is useless weeks after the last editing if not properly documented.

The documentation for the MAIN procedure is a little different that documentation for a sub procedure. I also include information on how the program is used and any other information required by the user.

Since this is the first source code that I am introducing, let me explain each basic section. These discussion will not be repeated in later sections.

2.5.2 Predocumentation

The page, title and name directives are used to identify the source code file. The name directive is ignored by later versions of MASM and is maintained only for compatibility. This directive used to define the module name referenced by the linker and was nice. Now the source code file name is used. I keep it because I am used to it, but it can be dropped.

2.5.3 Documentation

Each of the following sections explains my use of each part of the documentation section. I use the comment directive so that I do not have to start each line with a semicolon. This simplifies the editing process. I also use the division sign as the comment delimiter because I never have used it else where in the documentation. One of my future projects is to write a program that will print the documentation between the delimiters so that I can have just the documentation in a convenient reference form.

2.5.3.1 Header

I place the procedure name in all capitals with the version number at the extreme right margin. I also separate the line from the rest of the documentation with a row of asterisks. This is done to offset the documentation with a visible title.

2.5.3.2 Name

This section repeats the title directive in the predocumentation section.

2.5.3.3 Synopsis

In the Synopsis section, I place how to evoke the program including an explanation of all command line arguments. This is more for the programmer’s documentation vice being for the end user’s documentation.

2.5.3.4 Description

This section contains a more detailed explanation of the program, required input, output, etc. I include any other useful information for the user such as environmental variables used, DOS version dependency, required supporting files, etc.

2.5.3.5 Programming Notes

In this section, I include as a minimum the name and version of the assembler used, which switches were used, which memory model was used, which registers are used, and which processor instruction set was used. Then, I include any other information that I can think of that would be of use to me or any other reader in understanding how the MAIN procedure works.

2.5.3.6 Returns

Since the MAIN procedure can return to the startup code, this section does have meaning for the MAIN procedure. If the MAIN procedure returns to the startup code, it takes whatever is in the AL register and returns that value to the operating system. It is in this section, that I document what returns are possible and what they mean. If the MAIN procedure does not return, then so state.

2.5.3.7 Cautions

Any programming cautions about which the programmer should know are documented in this section. For the MAIN procedure, I include any cautions that would be of use to the user, such as the program requiring 512K of real memory to execute.

2.5.3.8 Memory Utilization

While this section is not as important for the MAIN procedure as for general purpose or library subprocedures, I have included it for completeness. When the project is near completion, i.e., no more coding changes, you can complete this section. For general-purpose subroutines, include information for all memory models supported for your most used instruction set like 286.

2.5.3.9 External Libraries

I include in this section, all calls to library routines by module name and library name. It may be useful to include versions of the module also. By default, I have included a stack overflow checking routine that I have provided as part of this package, see CheckStack on page 27 for a detailed discussion. Also note that I have commented this line using the traditional method. While this is not needed as this is nested in a comment directive, I do it to remind myself that this procedure is called only when DEBUG is defined.

2.5.3.10 External Procedures

This section contains all non library procedures called in the MAIN procedure. Generally these called procedures are unique to the project and along with the MAIN procedure, make up the project's source code. Included is the procedure's name and the source code filename in which the procedure is found.

2.5.3.11 Interrupts Called

Any interrupts used in this procedure are documented here by interrupt number, function number, subfunction number, and common descriptive name.

2.5.3.12 Global Names

All global names defined in the MAIN procedure are listed here. I relax this practice somewhat for the MAIN procedure. This procedure declares many variables used only in this section, while they may be public, I do not include them in this section. This section can help clear duplicate global name problems.

Generally, I create an include file that uses conditional assembly to determine what is done. If the label, MAIN, is defined, then that section in the include file is assembled where the variables are defined and storage allocated. Otherwise, the section that defines each variable as an `externdef` is assembled. Then, all I need to do is include this file in all source files, and I have a consistent interface in the source code, and all variables are available for addressing. When I add, modify or delete a variable, I do so only in one file. Lastly, using `externdef` ensures that only actual references are included in the .obj file so the size of the .obj file does not become abnormally large with references to unused variables.

2.5.3.13 Author

This section contains my name and copyright information. If it is your work, not copied, and you spent the time to write it, debug it, and publish it, then it should be copyrighted by you. This is true even if your program will be FreeWare. I recommend never releasing anything into the public domain. At a minimum, copyright your work and ask for no royalties but just acknowledgment of your code's use in the documentation.

2.5.3.14 History

I keep the version, date and comments in this section so as to refresh my memory on any changes that I have made over time.

2.5.4 Predata Information

Starting with this section of the template, I separate the code into blocks of directives and instructions that have a common purpose. I describe that purpose in a comment header. If required, I document each line of code to make clearer the logic flow. For a few files that I am providing detailed explanation of my code, I have labeled the blocks A, B, C, etc. for easy reference. I normally do not do this type of labeling in my source code.

2.5.4.1 Default Memory Model—Block A

Most of my assembly programming is done in the small memory model. So that I do not have to include the memmod define in the .mak file, I have the memory model default to small. This value is overridden by any value of memmod defined on MASM's command line or in the .mak file. Using the memmod define in the .mak file is a quick way the define the memory model. This serves the purpose of one location I have to change to change the entire project's memory model.

2.5.4.2 File Setup—Block B

This section contains the cats and dogs as it serves several purposes that all contribute to the overall structure of the code.

First, I have a include file that is used to define the processor directive. I use an include file so that I can change one file and reassemble all my source code for another instruction set. This is the easiest way I have found to keep the instruction set synchronized across all source code files.

Next, I use the .model directive to define the memory model and language. Note the use of the percent sign to ensure that the macro substitution of memory model occurs first. The memory model can be set on MASM's command line using the /D switch or in the .mak file.

I use the Fortran/Pascal/Basic calling protocol because it is more efficient for the called procedure to clear the stack of passed parameters than the calling function. The only limitation is that you can not use an variable list of arguments as you can in C. There also is a caution. The order of pushing arguments onto the stack is in the order they appear in the call statement. Since I use registers to pass arguments, in most cases, this does not bother me much.

Next, I place the assume for the ES register here since this assume is not set as part of the .model directive. While this directive should be in the code segment, it does work here and does not add to the clutter in the code segment.

I place the .dosseg directive to force the DOS segment order. This is required because the linker places some labels in the executable that is used by the Startup Code to determine the end of data. See Segment Order, page 13, for more information.

Lastly, there is another drawback if you make extensive use of uninitialized far data. It will make the .exe file quite large. See the discussion and work around on page 13.

2.5.4.3 Include Files—Block C

This block is reserved for all include files. There are two default include files in the template. The first one is startup.inc. If any global ASM Startup variables are referenced, then the startup.inc file must be included. If your MAIN procedure does not reference any of these variables, then this line can be deleted. Even if this line is not deleted your .obj or executable will not be increased in size. The second file is lib.inc. This file is mandatory for .exe file programs because it contains the directive to include the correct ALIB in the .obj file to link the correct startup code. For .com file program, this file is not needed for this purpose because you must ensure that the .obj from the desired startup code is the first .obj file to the linker.

There is another reason for including the lib.inc file. I keep `externdef`s to all general purpose library procedures in that file. So, if your source code contains any calls to any of these general purpose procedures available in ALIB, then this is the easiest way to include the references to them. This file serves a very similar function as header, .h, files in C.

2.5.5 Data Segments—Block D

This section is designed for your data. All variables that need to be initialized and written to are placed in the .data segment. Data, usually strings, that needs to be initialized but not written to are placed in the .const segment. Variables that do not need to be initialized but must be written to are placed in the .data? segment. This is done because, with an uninitialized stack, the linker places these segments last in the executable image and place information in the header telling DOS how much memory is needed for these segments. This method permits a small executable file because no actual space in the file is allocated for the uninitialized data in these segments.

Lastly, if more stack is needed than is provided in the startup code, add it here with the .stack directive. The .stack directive is commented out. Either delete it or uncomment and add stack. The startup routines allocate the following for stack space:

Version 0	512 byte	0.5 K
Version 1 and 2	1024 bytes	1.0K
Version 3	2048 bytes	2.0 Kbytes

2.5.6 Code Segments—Block E

This section contains the code. The first statement is an extern statement that specifies which startup version is required. The valid entries are:

Start_Up0
Start_Up1
Start_Up1s
Start_Up2
Start_Up2s
Start_Up3
Start_Up3s

2.5.7 Main Procedure—Block F

The MAIN procedure, passed arguments, and local variables are defined in this code block. Notice that the passed arguments are for linking to Start_Up3S. Just change the arguments to those below for other versions of the startup code.

Start_Up0	None
Start_Up1 or Start_Up2	ARGC:word, ARGV: ptr
Start_Up1s or Start_Up2s	ARGV: ptr, ARGC:word
Start_Up3	ARGC:word, ARGV: ptr, ENVR:ptr

Now, you can start coding your MAIN procedures.

2.5.8 Stack Check—Block H

The `Stack_Check` procedure call is placed in a conditional assembly block so that the call is only included in the debug version of your program. Just define the label, DEBUG. The call to this procedure is placed best after any local variables are defined. This should be the deepest a procedure uses the stack unless there is another call. Then, `Stack_Check` is called again.

EXECUTING ENVIRONMENT

3.1 .exe Startup Environment

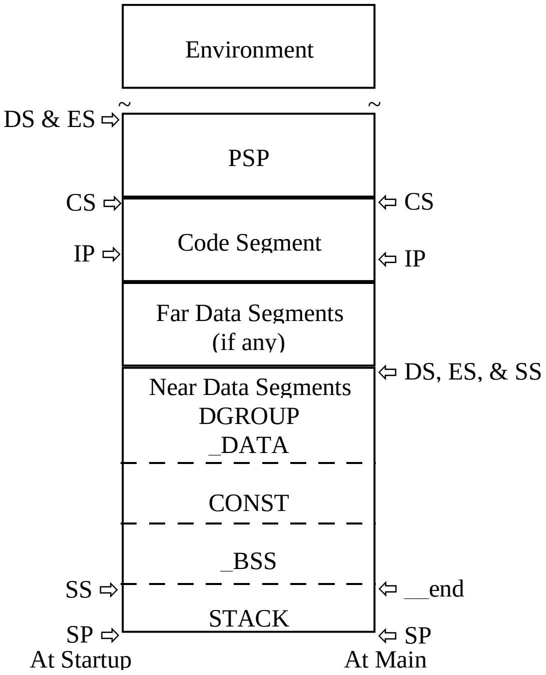

Figure 1 .exe Startup Environment

When an .exe program receives control, i.e., starts executing, the major registers are pointing as shown on the left side of Figure 1. This illustration is good for all memory models. The only difference is in the large code models, medium, large and huge, where the CS register does not have to point to the start of the code segment. The DS and ES registers point to the Program Segment Prefix, PSP (see Figure 1) even though the DGROUP directive is used or implied through the use of one of the canned memory models. Also while the stack is part of DGROUP, at startup, the SS register points to the STACK segment.

Generally, there is not any intervening memory between the environment and the PSP, but this does not have to be the case. Therefore, I have indicated in Figure 1 that there may be memory space between them.

The startup code initializes the value of DS and ES registers with the value of DGROUP. Then the SS register is set to point to the beginning of DGROUP, and the SP register is adjusted by adding the size of the _DATA, CONST and _BSS segments so that the SP register still will point to the current stack position.

This adjustment of the SS and SP registers allows any variable in DGROUP to be addressed as an offset from any of the three segment registers.

This means that all near pointers whether they were defined in the _DATA, CONST, _BSS, or stack can use any segment register. This is important so that pointers to automatic variables at runtime are equivalent to pointers to variables defined in the rest of DGROUP. Therefore, you as the programmer can use DS register as the segment register for any near pointer without worry as to whether it points to an automatic variable or a static variable.

Static variable positions that are known at link time remain constant throughout program execution. Automatic variables are created on the stack at runtime so their positions are only known during their existence. Lastly, if you have a recursive procedure, there is the possibility of multiple instances of the same automatic variable in existence simultaneously. The pointers to these automatic variables are all different, but can all use the same segment register. Again, you as the programmer do not have to worry about the type of variable nor when it was created.

Support of Large Data Models

I use the DOS segment order as shown in Figure 1 above. If you have very large uninitialized FARDATA, this situation could lead to very large executables because the linker will allocate space in the executable file for all the uninitialized data because initialized data in DGROUP is after the FARDATA uninitialized data in the file image. If FARDATA resides after DGROUP, the linker will not allocate space in the file image but increase the value of exMinAlloc in the .exe header. This value specifies the minimum amount of extra memory required by the program above the memory required to load the program image.

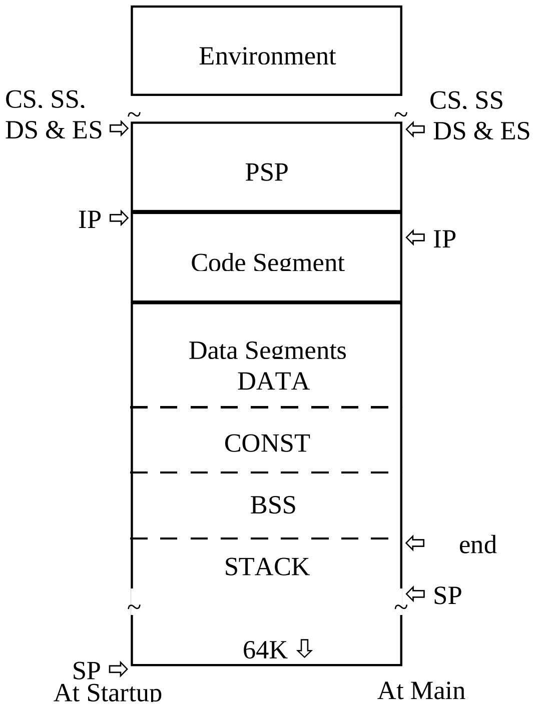

Figure 2 .com Startup Environment

Some programmers specify the order of segments and do not use the Microsoft default. Doing this will break my startup code. If you want to use my startup code and have uninitialized FARDATA, this is what you must do. Do not define this data but define structures that will allow you to address this data with pointers. Then in the MAIN or later procedure, allocate memory for this data. Use the returned segment value as the segment portion of the far pointer. The offset is the offset of the variable in the defined structure. There you have it.

3.2 .com Startup Environment

When an .com program receives control, i.e., starts executing, the major registers are pointing as shown on the left side of Figure 2. The IP register points to the first instruction at 100h. Lastly, if memory permits, the SP register points to 0fffeh or just below 64K if at least 64K of memory is available (see page 30 for a further discussion). All program segments are in one physical segment, called DGROUP. This is a much simpler environment than the .exe file format. All code and data are near.

Generally, there is not any intervening memory between the environment and the PSP, but this does not have to be the case. Therefore, I have indicated in Figure 2 that there may be space between them.

With .com file format, the stack extends from the end of the 64K boundary down to the end of the data. Generally, this is excessive. The startup code in this case only needs to move the SP register to a more reasonable location. The startup code default is to set the stack size to 512 bytes and place it just after the last of the data. Then all memory above the stack is released.

3.3 Segment Order

The segment order of the data segments is as shown in Figures 1 and 2. This is the DOS segment order. It is achieved by using the `.dosseg` directive in the startup code. The reason for using this directive is that it will minimize the executable image. As explained earlier, the reason is that the linker places initialized data, _BSS and stack segments, last in the image. The linker uses the minAlloc field in the .exe header to specify the memory required above the file image in memory for this unintialized data. In the .com file format, the linker just leaves that data out of the executable image. This practice has a vulnerability. See the section, Possible Problem with Available Memory and .com Files, page 30.

Another reason I use this directive is that the linker defines a label called `__end`. As shown in Figure 1, for .exe programs this label has the offset in DGROUP of the beginning of the Stack segment. The nice thing is that this `__end` is guaranteed to be paragraph aligned because the Stack segment is paragraph aligned. As shown in Figure 2, for .com programs this label has the offset in DGROUP of the first byte after the last data item in the _BSS segment. The trick here is that is not guaranteed to be paragraph aligned in this case. In this case, the startup code must add enough bytes to the `__end` offset so that it is paragraph aligned. Then this new offset can be used to shrink memory, see page 19, Combining the Stack into DGROUP, for a more detailed discussion on exactly how `__end` is used in the startup code.

3.4 Passed Arguments for Versions 1 and 2

Figure 3 to the left shows the stack with the arguments that are passed to the MAIN procedure from versions 1 or 2 of the startup code. The SP register is pointing into the stack at point A. The startup code calls MAIN and the return address is pushed onto the stack. The entry code in pushes the SP register and sets the BP register to the current value of the SP register. The important point here is that point A, the start of the passed arguments, while varies depending upon the code memory model, is at a know offset at assembly. Therefore, `*argv[]` and `argc` are at known offsets from the BP register. It just so happens that the pointers, `argv[]`, the null terminated array of pointers to the command line arguments, is just above `argc` on the stack, starting at point C. The first element is lowest in memory starting at point C, and the last element, the null, the highest in memory ending at point D. Lastly, the parsed and null terminated command line arguments are just above `argv[]`, starting at point D through point E. Notice hat the order of the arguments starts at low memory to high memory. The startup code parses the arguments from high memory to low memory so the order of the pointers is in the correct order.

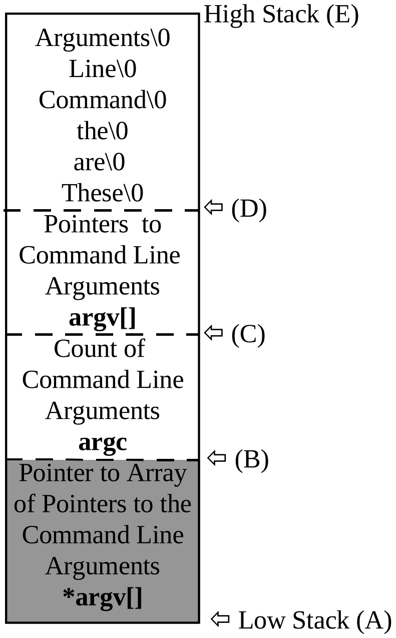

Figure 3 Version 1 & 2 Stack

You will notice in version 2 and 3, the drive/full path/filename and extension of the executing program is placed on the sack before the command line after the command line arguments are placed onto the stack so that the filename is in the correct position to become `argv[0]`.

As can be seen from Figure 3, `argv[]` also is at a known offset at assembly time. This fact and the added difficulty of addressing through an added layer of indirection lead me to the Simplified Addressing mode described on page page25. Notice also that `*argv[]`, the grayed area in Figure 3, is no longer needed as I address the elements directly as offsets from `argv[0]`.

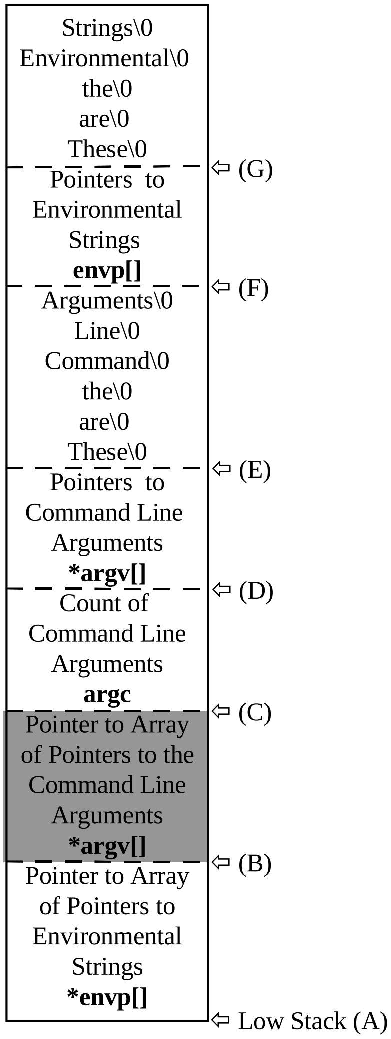
Strings\0
Environmental\0
the\0
are\0
These\0
Pointers to Environmental Strings
envp[]
Arguments\0
Line\0
Command\0
the\0
are\0
These\0
Pointers to Command Line Arguments
*argv[]
Count of Command Line Arguments
argc
Pointer to Array of Pointers to the Command Line Arguments
*argv[]
Pointer to Array of Pointers to Environmental Strings
*envp[]
⇦ (G)
⇦ (F)
⇦ (E)
⇦ (D)
⇦ (C)
⇦ (B)
⇦ Low Stack (A)

Figure 4 Version 3 Stack

3.5 Passed Arguments for Version 3

Figure 4 to the left shows the stack with the arguments that are passed to the MAIN procedure from version 3 of the startup code. It starts with the environmental strings parsed onto the stack followed by a null pointer terminated array of pointers to the environmental strings. These start at points G and F respectively. Then the stack looks just like with version 1 and 2. All the code for versions 1 and 2 will work here. The only difference is that the pointer to the environmental strings, position F, is not known at assembly time. This means that the `*envp[]` must be used so there is not any chance to use a technique such as simplified addressing as with `argv[]`.

Also note that the environmental strings are a copy of the environmental strings passed to the program. While they can be modified and used in the Load and Exec DOS Interrupt Function, there is no room for the strings to expand or add new strings. If you want to use a modified set of environmental strings, I recommend building an entirely new set.

3.6 Environmental Block Structure

An example of an environmental block expanded is illustrated in Figure 5, below. The format of Figure 5 above has the first line given the hexadecimal offset of the start of the memory paragraph. ASCII characters are represented as ASCII characters, while binary data is represented in byte length with the binary value in brackets, [].Note that the environmental block starts on a memory paragraph boundary because it is a separately allocated block of memory from its program code and data. The program code and data, starting with the PSP, normally is loaded right above the environmental string block with only a 16-byte DOS memory control block between them. If you have fragmented memory, e.g., a TSR that has released its environmental block, it is possible to have the program code and data block separated from its environmental string block.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
P	R	O	M	P	T	=	$	p	$	g	[0]	P	A	T	H
=	C	:	\	;	C	:	\	D	O	S	;	C	:	\	W
I	N	D	O	W	S	[0]	T	E	M	P	=	C	:	\	T
E	M	P	[0]	[0]	[1]	[0]	C	:	\	P	R	O	G	R	A
M	S	\	A	S	M	\	S	T	A	R	T	U	P	O	.
E	X	E	[0]	*	*	*	*	*	*	*	*	*	*	*	*

Figure 5 Environmental Block Structure

The format of the environmental string block is simple. Each environmental string is null terminated and is concatenated into one long string. The last environmental string is terminated with an extra null. This also can be thought of as the last string being a null string. Following the double null is a word variable that contains the binary count of the strings that follow. To date, only one string follows so this binary value always is one. This string is a null terminated string containing the drive, full path, filename and extension of the executing program. All the bytes from the end of this string to the end of the current paragraph of memory is undefined, shown as asterisks in Figure 5.

Note that this last string can provide the home directory of the executing program so that you do not have to relay on configuration files, user input, environmental strings or some similar technique to determine the home directory.

The format of Figure 5 above has the first line given the hexadecimal offset of the start of the memory paragraph. ASCII characters are represented as ASCII characters, while binary data is represented in byte length with the binary value in brackets, [].

STARTUP CODE—DETAILED EXPLANATION

4.0 General

In this section, I will give a detailed explanation of the startup code. There will be a full explanation of startup0.asm as the if the first version covered of the startup code. Since each subsequent version is based upon the previous, I will only discuss what is new in each version. Also, I will start after the documentation section as the pre-documentation and the documentation sections were covered in the ASM template discussion earlier and are self-explanatory.

4.1 Startup0.asm

Memory Model Specification—Block A

This is the first code in every procedure of mine. This code is there so that there is a default memory model. If I want the project to be assembled in the small memory model, I do not have to define the label, `memmod`, on the command line or in the project. Since most of y programs are written in this memory model, this is for my convenience.

Basic File Setup—Block B

I use this standard section to define the processor for which the code will be assembled. The processor specification is kept in a file so that I can change the processor specification in all project files by changing one line in one file and reassembling.

Next, I specify the memory model and language. I use the BASIC\PASCAL\FORTRAN calling protocol. Using this protocol requires that the passed parameters be pushed in left to right order and that the called procedure is responsible for cleaning up the stack. With MASM, this is easy to do as I just line up the parameters on the procedure statement, and MASM codes the `ret` instruction with the right size.

The third line specifies that the ES register is loaded with the DGROUP value. This allows me to use the ES register to reference any variables in DGROUG. This is important when the DS register must be loaded with another segment value other than DGROUP, say to support an Int 21h. After the interrupt, you will want to reset the DS register. As you will see, the startup code saves the segment value of DGROUP. With the ES register set to DGROUP, a simple `mov ds, DGROUP` instruction is all that is needed to reset the DS register.

The last line insures that the segment order is the DOS segment order. See section .exe Startup Environment, page 12, for more detailed discussion on the segment order.

Required Includes—Block C

This block specifies the required includes. The only include required is startup.inc. This file is required because I have not defined these startup variables in the source code. The `extendef`s are required to that the linker will include these variables from the .obj file in the appropriate .lib file. See page 22, Startup Supporting Code, for the reasons why I have used this technique.

Required Equates—Block D

This block specifies any required equates. The only one required in this file is `STACK_SIZE`. Remember, if you want to increase a projects stack size, do not do it by changing the number here. Increase the stack in the MAIN procedure source file, see page 11, Data Segments.

Data Segment Declaration—Block E

As described above, all data is defined external to the startup. Therefore the only needed in this code is the definition of `__end` label as external. This label is defined by the Microsoft Linker at the end of the data in the _BSS segment for the tiny memory model and at the start of the stack segment for all other memory models. This label is defined only when `.dosseg` directive is used. I use this label to determine the end of the data for releasing all the memory above the program.

Stack Segment Declaration—Block F

This section declares the nominal stack size. Any extra stack required by the application you are writing can be specified in your MAIN, see Data Segments, page 11. The size specified in the various source code modules are additive. For example, if 1 KBytes is specified in the startup code and you specify 1 KBytes in MAIN procedure, the total stack for program is 2 KBytes.

PSP Segment Declaration—Block G

This section defines the PSP segment and specifies the location of the Next Paragraph pointer and the Environmental pointer that DOS loads into the PSP. The Next Paragraph pointer is the segment address of the next block of memory above the program. This value can be used to determine the amount of memory available above the program. This section allows the startup code to reference this segment and the values that DOS stores in the PSP.

Code Segment Declaration—Block H

This section defines the start of the code segment and defines the proper relationship between the code segment of the startup code and that of the MAIN procedure. For small code models, the startup code external defines the MAIN procedure inside the .code segment of the startup code. For large code memory models, the extern definition is outside the .code segment of the startup code.

Tiny Model Origin—Block I

This code is in a conditional assembly statement. If the memory model is tiny, i.e., the output is going to be a .com file format, the origin of the code must be at 100h. That is the purpose of this code. If you are going to use this code for a .sys file, you will need to remove this code.

Startup Procedure Declaration—Block J

This code specifies the startup procedure. This procedure always is specified as FAR.

Initialization of Global Variables—Block K

This block initializes the global variables. Since DS points to the PSP, the DS register must be initialized with the DGROUP segment value for all memory models except Tiny. After this, all the global variables are initialized. Note that the value of NEXTPARA is the segment value of the next DOS memory block above or higher in memory when the program starts to execute. The startup code releases all the memory above the program before calling the MAIN procedure, so this value does not indicate the amount of memory owned by the application after MAIN is called.

This section best illustrates how I optimized the coding for .com programs. For tiny memory model, all segment registers point to the PSP. I have used conditional assembly to change the assume directives so that DS register points to the PSP when I need to address data there, and DGROUP when needed. This saves the segment overrides required in the other memory models. I first tried to redefine DGROUP for the tiny memory model to include the PSP. Unfortunately, when I did this, __end no longer is defined. This breaks my technique for determining the size of the program, so I used conditional assembly.

Combine the STACK into DGROUP—Block L

This block first determines the end of the data. In all memory models except for the Tiny memory model, the __end label is guaranteed to be paragraph aligned at the start of the STACK segment. This is not true for programs written in the Tiny memory model where the __end label is byte aligned to the end of the data. To ensure that the start of the STACK segment is paragraph aligned in the Tiny memory model, the following is done. First, 15 is added so that the position of the __end label so that it is equal or greater than the start of the next paragraph if the previous position was not paragraph aligned initially. Then, the resulting value is anded with 0fff0h that will truncate the value to a paragraph boundary. This value is saved as STACK_BOTTOM and can be used to determine if the STACK segment has been overwritten. If the model is TINY, the size of the default, as defined in Block D, is added to this value, else the value in the SP register is added. Lastly, the values of the SS and SP registers are set to the new values. Notice that the SS register is set first. Whenever the value of the SS register is changed, the next instruction can not be interrupted. This allows the SP register to be set without worrying about anything happening before the SS:SP register pair is set properly to the new values.

Releasing All Memory Above the Program—Block M

Remember that the value of the SP register is the byte offset to the last byte in DGROUP. In the Tiny memory model, this is the size of the program. Since Int 21h Function 4ah requires input in number of paragraphs, this value must be divided by 16. This is accomplished by bit shifting to convert the value to paragraphs. Also, the previous block made sure that the value was an even number of paragraphs, so that condition need not be checked again. In all other memory models, the size of the code and any far data segments and the size of the PSP must be added. Since this value is obtained by subtracting the segment value of the PSP from the segment value of DGROUP, this value already is in paragraphs. Lastly, DOS is called to resize the amount of memory to the size of the program.

Initializing the ES Segment Register—Block N

This is a personal preference of mine to have the ES register also point to DGROUP. The instruction in this block accomplished this. Most C compilers I have seen do not keep ES register loaded with a particular value. I do, and I maintain the value of DGROUP in the ES register. I do this because I find it convenient so that I can reload DGROUP after DOS calls where DS register is used, but the ES register is not. Remember that the SS register also points to DGROUP. If both ES and DS registers are modified, these registers can be reloaded using the SS register override, `mov ds, ss:DGRP`.

Calling the MAIN Procedure—Block O

Now, everything is set up to call the MAIN procedure that is called in this block.

Return Code—Block P

If the MAIN procedure returns, the value in the AL register is passed back to DOS as the return code using Int 21h Function 4ch. The last function of this block is to define the starting position of the program as the Startup Procedure.

4.2 Startup1.asm

PSP Segment Declaration—Block G

Starting with this version, two new variables are defined in the PSP Segment. They are the length of the command line and the command line itself.

Startup Procedure Declaration—Block J

This code differs from the startup0.asm in that the same code must support both the simplified and normal `*argv[]` addressing scheme. To allow both versions to coexist in a .lib file, the procedures must have different names. So that the code does not need to be changed, I have included an `equate` that will change the procedures name if `SIMPLE` is defined. Notice that every subsequent line that uses the procedure name starts with a percent sign. This percent sign tells the assembler to make all the text substitutions prior to assembly of that line. This ensures that the proper procedure name is used.

Determine Presence of Command Line Arguments—Block N

This code checks `PARM_LEN` in the PSP to determine if there are any command line arguments. If there are not any, this code block pushes a null integer for `argc` and a null pointer for `*argv[]`. That is four or six null bytes depending on the memory model. While not specifically mentioned, the simplified model also is supported. When in the simplified addressing model, the order of arguments is reversed, since they are both nulls, effectively it is the same.

Moving the Command Line onto the Stack—Block O

This is a three-step process. First the length of the command line must be determined. The length is stored in the variable that I call `PARM_LEN` in the PSP. To ensure that it is an even number, I add one and then drop off the last bit. If an extra byte is included, it is the terminating carriage return which is not included in the count. The second step is to subtract this length from the SP register to make room on the stack for the command line. Lastly, the command line is copied onto the stack.

Parsing the Command Line and Building `*argv[]`—Block P

This next section is where the command line is scanned for spaces. Spaces are converted to nulls unless the spaces are between double quotes. This feature allows the user to enter an entire string as one argument. The double quotes themselves are replaced with nulls so you as the programmer do not have to worry about handling these delimiters. As I work down the command line from end to front, whenever the code was in a word and found a spot where a null was to be stored; the code pushes the previous address onto the stack. Notice that if the memory model is one of the large data models, the stack segment also is pushed. This is how `*argv[]` is built. Also, during this time, a count of command line arguments is kept. The value will be saved in the next block as `argc`.

Creating `argc` and `*argv[]`—Block Q

If SIMPLE was defined, then only the argument count is pushed as `argc`. If SIMPLE is not defined, a pointer to the `*argv[]` is pushed after `argc`.

4.3 Startup2.asm

Adding the Filename as `agrv[0]`—Block P

The difference between version two and version one is that version two provides the drive, full path, filename and extension as `*argv[0]` similar to the C language. The filename is found at the end of the segment containing the environmental strings. See page 15 for a discussion on the structure of the environmental segment. I scan for two null bytes which signals the end of the environmental strings. I then move four bytes down from that first byte of the double null. This is the beginning of the filespec. I determine the length. Lastly, I make room for it on the stack and copy it onto the stack using a technique similar to that I use for the command line arguments.

This adding of the filespec is done after copying the command line arguments so that the filespec is the last or at the lowest address on the stack. Since I parse the stack from high address to low address, the filespec address will be the last pushed onto the stack. The pointer with the lowest address is `*argv[0]`.

4.4 Startup3.asm

Determine the Size of the Environmental Strings—Block M

Since the size of the environmental variables can be quite large, I increase the size of the stack segment by the size of the environment. This block determines the size of the environment. The last environmental string has a second null terminating it. I search the environment for that second null. To transfer an even number of bytes, I and the length with 0fffeh. If the original length was odd, only the last null is lost. The size the stack must be increased is an even multiple of paragraphs, so I add 15 and truncate again using 0fff0h. This value is used in the next section to adjust the size of the program.

Adjust Size of Program and Ensure that DGROUP Does Not Exceed 64K—Block N

The sized determined in Block M is added to the to of the stack. If an overflow occurs, DGROUP would exceed 64K. In this case, an error message is sent to STDERR and the program is terminated with a return code of 80h. Second, there now is the possibility of the size of the program increasing. Therefore, I check to see if the request to resize the program failed. If it did fail, another error message is displayed to STDERR and the program is terminated with a return code of 81h.

Copy Environmental Strings Onto the Stack—Block O

This section makes room on the stack for the environmental strings by adjusting the SP register. Then the environmental strings are copied onto the stack.

Build `*envp[]`—Block P

I basically use the same algorithm in parsing the environmental strings as the command line but these strings already are null terminated, and I do not have to check for double quote delimiter. Therefore, I only have to check for a null to determine the start of string. I push the start of every string onto the stack. Again, since I am working from the end to the start, the pointers are in the correct order. The address of the last pointer pushed, `envp[0]`, which is `*envp[]` is saved in the DX register.

Creating `*envp[]`, `*argv[]` and `argc`—Blocks V and W

Depending whether SIMPLE is defined, the order of placing these values on the stack are:
SIMPLE defined:
`argc` and `*envp[]`.
SIMPLE not defined:
`argc`, `*argv[]` and `*envp[]`.

4.5 Startup Supporting Code

There are two support .asm files. These files are needed so that all the startup procedures can coexist in the same .lib file. The problem is that all the startup procedures use the same variables that are publicly defined. When building the .lib file, only the first startup code added to the .lib will have these publicly defined. All subsequent startup .obj files added will not have these variable listed as public because they were defined previously. If this condition was allowed to exist, the linker would add the correct startup procedure based upon the startup procedure's name. If this startup procedure does not have the publicly listed variable names, the linker will search the .lib for the procedure that does have those names defined. The second startup code will be included. Now two programming starting points are defined. The linker will terminate with an error message.

The solution is to define these variables in a separate file so that these names are defined only once in the .lib file. Now the linker will include this file because the name is defined as external in the startup code. The name of this supporting code file is SUDATA.ASM.

Since one of these variables, `ENV_STR_LEN`, is only used in Version 3I have a separate file just to define this variable. The name of this supporting code file is SUDATA3.ASM. This variable just as easily could have been defined in STARTUP3.ASM. I have defined this variable this way for consistency and possible future growth.

4.6 Maintaining the .lib Files

Maintaining the .lib files is quite easy. Just compile the procedure for each memory model and add them to the appropriate .lib file. I use a .bat file to do this. All the files use their filenames. The only problem is that for versions 1 through 3 of the startup code have two flavors, normal and simplified `argv[]` addressing. For this to happen, I needed to change the name of the startup file when assembling and adding to the simplified addressing variation to the .lib file. I used the filename, `strtpXs`, where `X` is the version number. If you want to update one of these .obj modules, you will have to use this naming scheme for the filename. Changing the filename does not affect the linker finding the correct version to load to create the .exe file. Linking relies only upon the procedure filename which is not limited to 8 characters.

DEMONSTRATION PROGRAMS

5.0 General

Included in the startup code are three demonstration programs. These programs were written to show how to use the startup code. Simply, they display the global variables set, the memory model used to assemble the program, and the information particular to the version such as the command line arguments and environmental strings.

At the heart of the demonstration programs is the MAIN procedure. These MAIN procedures follow the template very closely, but there are significant differences. While your MAIN procedures will be targeted at one memory model, these procedures were written to be assembled in any memory model. I have done this to demonstrate how to address the various elements of the startup code in any memory model.

5.1 Demo0.asm

This MAIN procedure was designed for version 0 of the startup code. The code is straight forward as only the global variables are displayed as version 0 does not pass any command line arguments nor any environmental strings. The entire program consists of displaying a string telling the user what is going to be displayed and then displaying the value.

First, notice the use of lib.inc and startup.inc. These include files take care of defining global variables and procedures. It is very similar to writing c code. If you use the correct header file, you can reference external variables and procedures.

The only code in MAIN0.ASM that is memory model dependent is found in Block E. This code sets up the relationship between the MAIN procedure and the startup procedure. Normally, you as the programmer would choose the memory model and hard code this section to that memory model. Then the conditional assembly would not be needed.

5.2 Demo12.asm

This is the MAIN procedure for versions 1 and 2 of the startup code. Since these versions differ only in that version 2 provides the program name as `*argv[0]`, the same MAIN procedure can be used for both versions of this startup code.

To use DEMO12.ASM, first select which version of the startup code you will be using. Place the desired version of the startup code in the `equate` and the `extern` directive statements in Block F. This will ensure that the proper version of startup code is linked.

There is another choice that must be made when using versions 1 through 3 of the startup code. This is the choice whether simplified argument addressing will be used. This affects the order of arguments as they appear on the MAIN procedure declaration. I will cover the differences between these two variations in a more detail shortly. The point here is that you need to make two changes to support either addressing mode. To simplify this process, I have placed the code into conditional assembly statements so that all you need to do is to define the Label SIMPLE which can be done within the Programmer's Workbench or on the command line using the /D option. The only thing that must be done is to indicate the version of the startup code by using 1 or 2 to indicate the version in the `define` and the `extern` directives. Everything is now done.

As for the code of DEMO12.ASM, it is based upon DEMO0.ASM. The only changes are around the MAIN procedure and the code were the command line arguments are displayed. The actual code was just a loop that displays each command argument on its own line. The most important feature of this code is to illustrate `*argv[]` addressing.

5.3 `*argv[]` Addressing

See Stack Structure, page 14, to learn the difference between the normal and simplified addressing as to how they differ as to the placement of the parameters on the stack. In this section, I will describe how to use parameters using either addressing mode. This is illustrated in Block X in the code. Again, since this code is to illustrate both addressing types, I have used conditional assembly statements. Since I am addressing an element of an array, either addressing method requires the calculation of an offset into the array. The first two instructions calculate that offset. Notice that the index must be shifted once for word pointers, small data memory models, and shifted twice for doubleword pointers, for large data memory models. To handle this difference I define the constant ShiftCount using conditional assembly in Block F.

5.3.1 Normal Addressing

The first section of assembly statements illustrates how to address the command line arguments using the normal addressing scheme. To load the pointer to a specific string is a two step process. The first step is to load the pointer to the array of pointers. This is achieved with the

```
     mov  bx, ARGV     ; for small data models
```
or
```
     les  bx, ARGV     ; for large data models.
```

I used the LES instruction as I wanted the segment value in the ES register. Since pointers are memory model dependent, I have written a macro that I define in Block D that will expand to the correct code depending on the memory model. This allows me to change the memory model after the code is written without having to go searching through source code and make changes.

The second instruction is a move to move the offset into the desired register. Note that I have the offset calculated and stored in the DI register. Also note that this instruction will work with both large and small data models. The reason is that I know that the command line arguments are on the stack that is part of DGROUP, so I do not lead to reload the segment value. Technically, I only needed to load the offset vice the whole pointer in the first instruction, but I wanted to show how to write memory model independent code.

5.3.2 Simplified Addressing

In the MAIN procedure, this addressing scheme is much easier. To load a pointer, only one instruction is needed as illustrated in the second part of the conditional assembly in block X. Since I am passing this pointer to another procedure, I need to use the macro so that in large data models, the segment value also is passed.

5.4 Demo3.asm

This is the last demonstration program. It is based upon Demo12.asm. First, the displaying of the variable, `ENV_STR_LEN`, is added in block Q. Next, I modified the `*argv[]` displaying routine for displaying the environmental strings. Here, I could not use simplified addressing so I took that out.

OTHER PROCEDURES AND SUPPORT FILES

6.0 Background

Included with the STARTUP code are five other procedures. These procedures are general purpose procedures used in the DEMO program to check for stack overflow , convert number bases and display strings. I will not explain the algorithms used, but, like all my source code, these procedures are well documented and should be self-explanatory. The source code fro these procedures can be found in the `\libsrc` directory. Also, these procedures are good examples on how to write procedures to be assembled and linked in any memory model. The .obj modules for these procedures for all memory models from tiny to huge are in the respective .lib files in the `\libs` directory.

These procedures are:

STCKCHCK	Checks for stack overflow.
BIN2HEX	Converts a binary word into a hexadecimal ASCIIZ string equivalent.
PRINT	Displays an ASCIIZ string to stdout.
PUTS	Displays an ASCIIZ string to stdout and then sends a CR and LF to stdout.
UTOA	Converts an unsigned binary word into an ASCIIZ string equivalent.

Also included are several support files, all include files with the .inc extension, that support the use of the startup code and assembly language programming. These files are found in the \include directory. The files are:

LIB	`Externdefs` for all library procedures and defines proper library for the memory model.
PROCESSOR	Defines the processor instruction set for a project.
STARTUP	`Externdefs` for all variables defined in the Startup Code.

Lastly, I have included four batch files for the maintenance of the .lib files. They are `SU.BAT`, `SUS.BAT`, `MAKELIBS.BAT` and `RMLIBS.BAT`.

6.1 StckChck.asm

This procedure is a special purpose procedure that checks for stack overflow. This procedure was designed to be used while debugging. This procedure takes the stack bottom, which is saved by the startup code, adds a safety margin defined in the procedure and compares this calculated offset with the current SP register value. If the value in the SP register is equal or less than this calculated value, an error message is displayed to STDERR.

Since this procedure was designed to continue executing even if an overflow is found, I recommend setting a break point in the error handling code. This will allow you to trace back through the calls that led to this procedure being called and determine why so much stack is used. If there was not a programming error that led to so much stack being used, then you need to increase the stack size for the program. This is should be done in the MAIN procedure. See Data Segments, page 11, for how to increase the stack.

Also see p 11 for how to include this procedure in your code. I only include this procedure in the project specific modules. As you can see, I have not included it in the debugged library procedures. Note that in the library procedures I include the stack size used in the documentation

6.2 Other Procedures

Bin2Hex

This is a general purpose procedure that converts a binary word into a hexadecimal ASCIIZ string.

Print.asm

This is a general purpose procedure that prints to stdout the passed ASCIIZ string that is pointed to by DS:DX.

Puts.asm

This is a general purpose procedure that prints to stdout the passed ASCIIZ string that is pointed to by DS:DX followed by a CR/LF pair so the cursor will be at the start of the next line.

UToA.asm

This is a general purpose procedure that converts an unsigned binary word into a decimal ASCIIZ string.

6.3 Include Files

Lib.inc

This is an include file that supports the use of procedure libraries in assembly language. The first part of the include file uses the memory model and conditional assembly to include the appropriate library. Then, the second part has the `externdefs` for all procedures in the libraries. I use conditional assembly here to establish the correct relationship between the .code segment and the procedures based upon the code memory model. Whenever I add a procedure to the library, I add the `externdefs` in this file. This file serves the same purpose as header files do in the c language.

Procesor.inc

I use this include file to define the instruction set to be used in a project. I copy this include file into the project directory where this copy will be used before the one in the include directory. I then change the target processor directive. I find that if I do need to change the instruction set, all I need to do is change this one file and reassemble the entire project. Note that the code to include this file is in the templates included in the startup code so that using it is built in.

Startup.inc

This include file has all the `externdefs` for the variables defined in the startup code. This include file is included by default in the MAIN procedure source file as it is part of the template, see p 10.

6.4 Batch Files

There are four batch files included to assist in the maintenance of the .lib files. These files automate the adding, updating and removing object modules from all the .lib files. The syntax of their use is in the documentation embedded at the start of each batch file. Their uses are:

Su.bat

This batch file is used to add or update the startup procedures that use the normal addressing scheme for `argv[]` to the .lib files. This batch file assembles the specified startup procedure in one memory model, then adds our updates the appropriate .lib file. The same procedure is performed for each memory model except tiny. A list file for each .lib file is produced so that you can review the contents of each .lib file.

Sus.bat

This batch file is the same as `SU.BAT` except the procedures are assembled using the simplified `argv[]` addressing. Therefore, only versions 1 and higher are assembled using this batch file. Also remember to change the name of the source file. I used strtp with the version number to differentiate them from the normal addressing versions.

Makelibs.bat

This is the basic batch file to add or update object modules in the .lib files. This is he batch file used to add or update the two startup support files, sudata.asm and sudata3.asm.

Rmlibs.bat

This is the basic batch file to remove object modules from the .lib files.

.COM FILE INFORMATION

7.1 Writing Modular .com File Programs

When I started programming in assembly language in the mid 80s, most assembly programs were written in the .com file format and were written with all data and code in one segment in one file. An outline of this style would be:

```
PROGSEG  segment
    org  100h
START:   jmp  REAL_START
;
; Data went here
;
REAL_START:
;
; Code went here
;
PROGSEG  ends
    end  START
```

This style was a result of the early linkers that could not output .com file program directly and assemblers that could not handle forward references. As linkers and assemblers improved, the requirement to write code in this manner disappeared.

Now, the real requirement is that all the code and data must reside in one physical 64K segment. You can write .com file programs with as many program segments as the linker can handle. All you have to do is to place all of them in one group. If you use the Tiny memory model, this is done for you because the _TEXT segment that contains the code is included in DGROUP. Just look at the demonstration programs where I create .com format executable that even can use procedures in .lib files. Notice also, you can link to code in lib files in the Tiny memory model. Therefore, you do not have to change your style just because, you want your final program in the .com file format.

7.2 Possible Problem With Available Memory And The Loading and Executing of .com Files

Since there is not any header with a .com file, the DOS loader does not know how much memory is required by the program beyond the end of the file image. With the latest linkers, uninitialized data along with the stack is missing. The DOS loader by default allocates all available memory to the .com file program and sets the SP register to 0ffffh +1 and pushes a null word. This results with the .com program starting with the SP register set to 0fffeh. This usually results with the stack growing down from the 64K boundary with more than enough room for uninitialized variables past the file image loaded into memory.

The problem arises when there are less than 64K of memory available to load and execute a .com file program. The MS-DOS Encyclopedia[1], that covers up to DOS 3.2, states that the DOS loader will set the

[1] Ray Duncan, editor, The MS-DOS Encyclopedia, Microsoft Press, Redmond, WA, 1988, p. 143

SP register to one byte past the highest offset in available memory owned by the program. After the null is pushed, the SP register is pointing into initialized data. Obviously, it is easy to see that this is a formula for a catastrophe. To illustrate, let's have a program image of 6K with 10K of uninitialized data. Now, if only 8K is available, DOS will load, and the stack and program will start to execute. Data will overwrite the stack and beyond. Overwriting the stack will lead to a program and,, probably, a system crash.

The MS-DOS Programmer's Reference, Version 5.0[2], states that there must be room for the PSP, the program image and 256 bytes. If this amount of memory is not available, the loader will fail with an insufficient memory. While this is better, it does not solve the basic problem.

It is my recommendation to use the small memory model vice the tiny memory model. This way, the actions of DOS are known. Now, when would this situation possibly arise. I have written many programs that I use from within Microsoft's Programmer's Workbench. How much memory is available when the Workbench shells out? I do not want to leave it up to chance.

[2] MS-DOS Programmer's Reference, Version 5.0, Microsoft Press, Redmond, WA, 1991, p. 75

PROGRAMMING INFORMATION

8.0 Writing code to Support All Memory Models

Generally, there is just a little extra effort in writing code that will assemble in all memory models. You will not write all assembly language procedures in this manner. Only those procedures designed o be general purpose library procedures will be written this way. I wrote HEX2BIN, PUT, PUTS and UTOA this way. Generally, your project specific code is written in a single memory model.

Memory models can be broken down to being to either large or small code and large or small data thusly:

Memory Model	Code Model	Data Model
Tiny	Small/Near/Offset only	Small/Near/Offset only
Small	Small/Near/Offset only	Small/Near/Offset only
Compact	Small/Near/Offset only	Large/Far/Segment & Offset
Medium	Large/Far/Segment & Offset	Small/Near/Offset only
Large	Large/Far/Segment & Offset	Large/Far/Segment & Offset
Huge	Large/Far/Segment & Offset	Large/Far/Segment & Offset

In the above table, small and large used in the Code and Data Models is not the same as Small and Large in the Memory Model column. It is quite common to read small code models. When small is used in conjunction with code or data models, the small refers to the use of near pointers that are 16-bits in size and consist of the offset value only. Large in the same context refers to the use of far pointers that are 32-bits in size and consist of both the segment and offset values.

In most cases, the code memory models handle themselves. The call and return instructions will be assembled according to the defined memory model. If a consistent memory model is used through out the project, the assembler will produce the correct entry and exit code without any programmer intervention. Unless you are working with pointers to procedures, the coding of call and the returns, whether near or far, is transparent to you as the programmer. If you need to use pointers to procedures, see the next few paragraphs as you can use the same techniques here as you do for large data models.

Changing from a small data model to a large data model will affect the code. Through the use of macros and a little care, the problem can be minimized.

First, small data memory models use near pointers that are word length. The usual registers used are BX, SI and DI registers. To load the pointer, use the simple `mov` instruction. The default segment register used by these registers is the DS register. This is the reason that it is so important for the stack being combined into the DGROUP so that static and automatic variables can be addressed using the same segment register.

Far pointers are a little trickier. Loading far or doubleword pointers uses the `lds`, `les`, `lfs` and `lgs` instructions. Here is where macros come into play to make this coding easier. To make the code memory model independent, conditional assembly must be used. Instead of cluttering up your code with conditional assembly, use the below macro. Using a macro saves time as you only have to write the code only once. The basic format for all of the above instructions is:

```
inst     destination, source
```

that is exactly the same format as the `mov` instruction used in the small data models. Therefore, a macro is easy to write that will handle both cases. You actually will need four, one for each of the above instructions. Once you have one, all you have to do is change two letters, and you have another. Now for a little style I use when writing macros. To make a macro easily identifiable, the first character is an at sign, @, and the macro name is all in upper case. I use all lower case for instructions so my macros are readily recognizable. For `lds`, my macro is:

```
@LDS     macro    dest, sour
if @DataSize
    lds  dest, sour
else
    mov  dest, sour
endif
endm
```

Note that the value of @DataSize is 0 or false for small data models so the `mov` instruction is encoded. The value is 1 or true for large memory models so the `lds` instruction is encoded. Note that the value is 2 for huge memory model, but this value works just like the value of 1. This construct makes conditional assembly quite easy.

Now, I just copy and paste this macro three more times and substitute in the correct middle letter, and I have a full set of macros to handle all situations. I place these in my basic macro library file with I can include when I want to use them.

That takes care of loading far pointers; what is left is the storing of far pointers. This operation is not as conducive to the use of macros because the source code can be quite varied. Generally, I code the two instructions manually each time. Since I always declare far pointers as doublewords, the code would be:

```
FarPointer   dd   ?

mov      word ptr FarPointer, ax
mov      word ptr FarPointer + 2, dx
```

FUTURE IMPROVEMENTS

9.0 Future Improvements

There are two areas that I can see improvements. These are setting up a near heap and adapting the startup code to work a console application under Windows 95 and NT.

9.1 Near Heap

The first improvement is to have the startup code create and initialize a near heap. This would be accomplished by expanding DGROUP to 64K. The stack would be moved to the high end of DGROUP. The near heap would be from the end of the uninitialized data to the bottom of the stack. Modifications to the startup code would be fairly minor. These would be to determine the size of DGROUP, 64K or what is available, move the stack to the top, and finally set up a memory control block for the remaining memory between the end of data and the stack. To implement a near stack, three procedures are required: an allocation procedure, a deallocation procedure, and a clean up procedure that will combine consecutive free memory blocks when there is insufficient memory to satisfy a request.

The value of a near heap is that it would provide the capability to support linked lists, etc. Again, this would allow pointers to the linked lists be near.

I have not implemented this capability only because I have not needed this capability. The likelihood is slim that I will.

9.2 Windows 95 and NT Support

The last improvement is to modify the startup code to support being a Windows 95 or NT console application. The code will need to be converted to the flat memory mode. I do not see any future in writing Graphical User Interface applications in assembly language, but I do see a future for writing console applications in assembly. This modification probably will occur sometime in the future.

REGISTRATION

Registration Information

Complete the following form to registry you copy of STARTUP. With registration you will receive the current version, if you registered an earlier version, or else the next upgrade for free. You also have the option to have a printed User’s Guide sent to you for a small extra charge. Remember that you must register STARTUP if you use this program for more than the 30 days trial period.

Noncommercial Registration	
Basic	$10.00
Allows use on one computer at a time for noncommercial use. *Programs developed using this license can not be distributed whether commercially, by shareware or by freeware.*	
Commercial Registration	
Basic	$20.00
Allows for use on up to ten stand alone, networked, or any combination of computers on a network for commercial use.	
Each computer above ten	$1.00
Programs developed using this license can be distributed as long as the MoonWare Shareware copyright appears in the documentation. Source code can not be distributed without the prior written agreement of the author.	
Enterprise Registration	Contact the author

Mail the registration to:

Raymond Moon
16005 Pointer Ridge Drive
Bowie, MD 20716-1744

STARTUP Registration Form

Name: ____

Company: ____

Address: ____

____

City, State & Zip: ____

e-mail Address: ____

Registration

Basic Non-Commercial ($10.00) $_______________

Basic Commercial ($20.00) $_______________

_______ Additional Commercial Computers ($1.00 each) $_______________

TOTAL: $_______________

Comments: