General Notice

When using this document, keep the following in mind:

1. This document is confidential. By accepting this document you acknowledge that you are bound by the terms set forth in the non-disclosure and confidentiality agreement signed separately and /in the possession of SEGA. If you have not signed such a non-disclosure agreement, please contact SEGA immediately and return this document to SEGA.

2. This document may include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new versions of the document. SEGA may make improvements and/or changes in the product(s) and/or the program(s) described in this document at any time.

3. No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without SEGA'S written permission. Request for copies of this document and for technical information about SEGA products must be made to your authorized SEGA Technical Services representative.

4. No license is granted by implication or otherwise under any patents, copyrights, trademarks, or other intellectual property rights of SEGA Enterprises, Ltd., SEGA of America, Inc., or any third party.

5. Software, circuitry, and other examples described herein are meant merely to indicate the characteristics and performance of SEGA's products. SEGA assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples describe herein.

6. It is possible that this document may contain reference to, or information about, SEGA products (development hardware/software) or services that are not provided in countries other than Japan. Such references/information must not be construed to mean that SEGA intends to provide such SEGA products or services in countries other than Japan. Any reference of a SEGA licensed product/program in this document is not intended to state or simply that you can use only SEGA's licensed products/programs. Any functionally equivalent hardware/software can be used instead.

7. SEGA will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's equipment, or programs according to this document.

NOTE: A reader's comment/correction form is provided with this document. Please address comments to :

SEGA of America, Inc., Developer Technical Support (att. Evelyn Merritt)
150 Shoreline Drive, Redwood City, CA 94065

SEGA may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

SEGA OF AMERICA, INC.
Consumer Products Division

Sample Game Program
User's Manual

Doc. # ST-159-R1-092994

READER CORRECTION/COMMENT SHEET

Keep us updated!

If you should come across any incorrect or outdated information while reading through the attached document, or come up with any questions or comments, please let us know so that we can make the required changes in subsequent revisions. Simply fill out all information below and return this form to the Developer Technical Support Manager at the address below. Please make more copies of this form if more space is needed. Thank you.

General Information:

Your Name _____ **Phone** _____

Document number ST-159-R1-092994 **Date** _____

Document name Sample Game Program User's Manual

Corrections:

Chpt.	pg. #	Correction

Questions/comments: _____

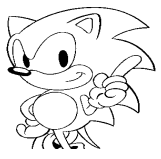
Where to send your corrections:

Fax: (415) 802-1717
Attn: Evelyn Merritt,
Developer Technical Support

Mail: SEGA OF AMERICA
Attn: Evelyn Merritt,
Developer Technical Support
150 Shoreline Dr.
Redwood City, CA 94065

Sample Game Program
User's Manual

(This page is blank in the original Japanese document.)



1. Introduction

To develop a game, responsible people are needed for planning, programming, designing, sound development, etc. The following three items are among the first things a programmer must keep in mind:

1) Characteristics of the game machine hardware.
2) Development environment of the game machine.
3) Game programming basics

This manual concentrates on explaining "(3) Game Programming Basics," using sample programs. To aid in understanding this manual, it would be beneficial to play several games on a MEGA Drive or other game machine. Also, it would help to read the coding regulations from the sample program list.

The program must be changed since part of it does not satisfy the game creation standard. For instance, the program performs input of the controller by the through mode, but this should be changed because the SMPC command must be used.

2. Overview

This sample program is located in the SEGAGAME directory. The sample program structure is shown in Figure 1; however, this manual breaks it up into the following classifications.

Game Sequence

These are display, Sega logo display, title display and program select in Figure 1. These are common for most games.

Basic Function Program

The basic function programs are the sprite samples and scroll sample shown in Figure 1.

Action Control

This is the 2D shooting game and a game demo is shown in Figure 1. The manual explains how to create the action game in this section. The program that expresses the action including the player (own machine), enemy (other machine), (collision check), background scroll, etc., is also covered. This action is controlled by the action control program.

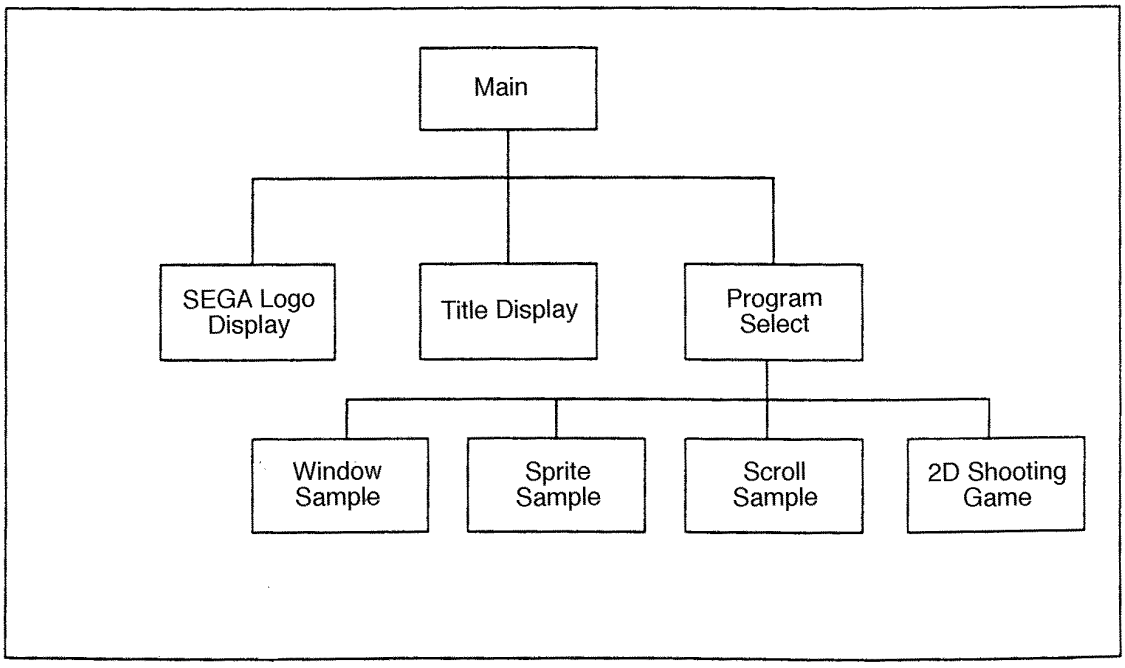
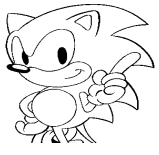


Figure 1 Program Structure



3. Game Sequence

Figure 2 shows the game sequence in flow chart form. This flow is controlled within the program by status flags called game mode variables.

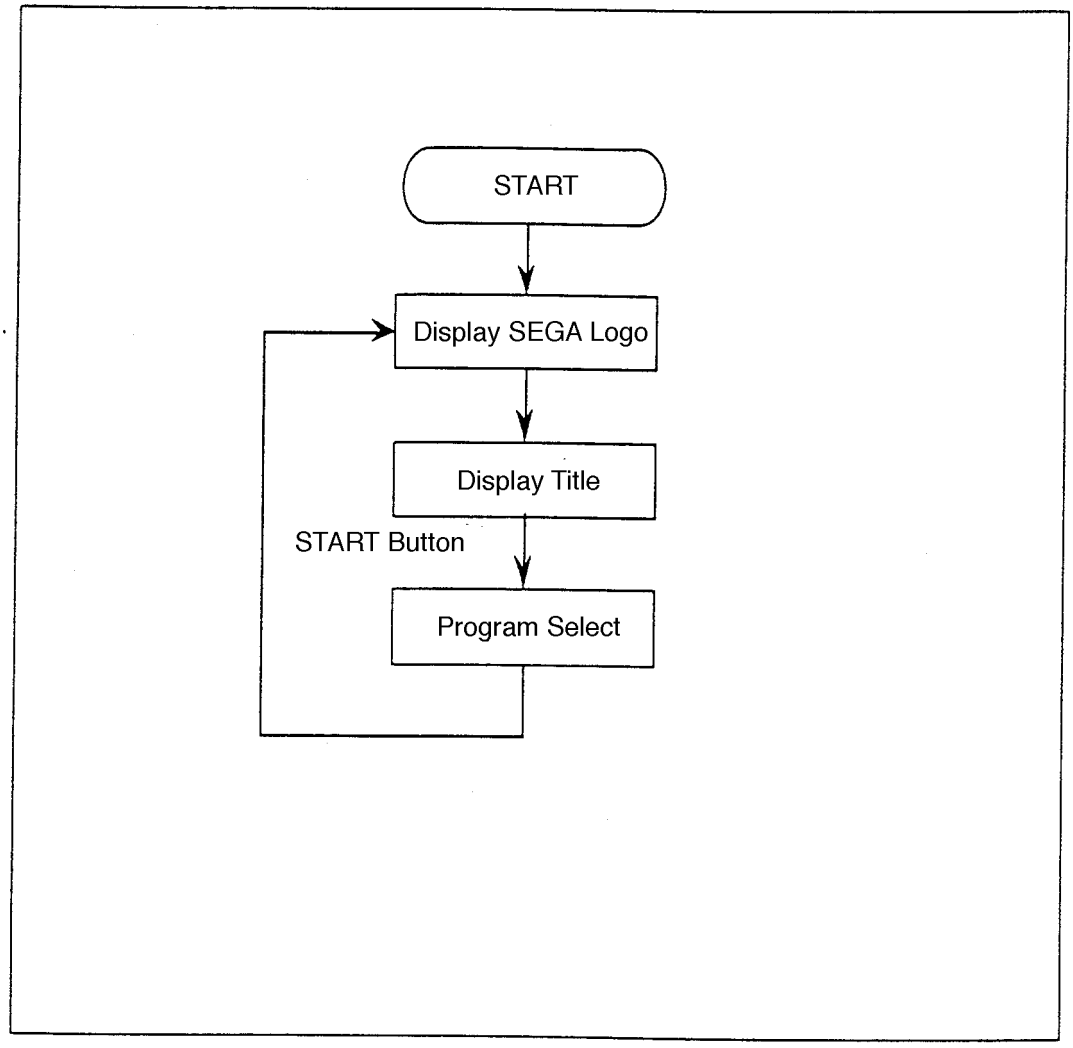
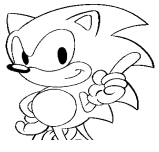


Figure 2 Game Sequence Flow

The game sequence uses C language main function and appears like the following.

```
Uint32 SMMA_MainMode;                /* Game Mode Variable      */
Uint32 SMMA_Mainlevel;               /* Level in the Game Mode  */

void main(void)
{
      SMMA_IniSystem();              /* System Initialization   */
      SMVl_SprCmdStart();
      SMVl_SprCmdEnd();
      SCL_DisplayFrame();
      SMMA_MainMode = LOGO_MODE;
      SMMA_Mainlevel = 0;

      for(;;) {                      /* Infinite Loop           */
            switch(SMMA_MainMode){   /* Game Mode?              */
            case LOGO_MODE:          /* At Sega Logo State      */
               SMLO_SegaLogo();      /* Display Sega Logo       */
               break;
            case TITLE_MODE:         /* At Title State          */
               SMTI_Title();         /* Display Title           */
               break;
            case SELECT_MODE:        /* At Program Select State */
                  SMSL_Select();     /* Execute Program Select  */
                  break;
            }
      }
}
```

Depending on the game mode variables in the main function, different subroutine functions are called up and used repeatedly. The game mode variable determines the following operation states. Because the game mode variable is defined as an external variable, the status can be changed from any of the subroutines. The LOGO_MODE is held as the initialization value, so start the system at the SEGA Logo display. The contents of the various subroutine processes are shown below.



SMMA_IniSystem() Initialize

Initializes the system as shown in Figure 3. Initialize the hardware, etc. within the system initialization process.

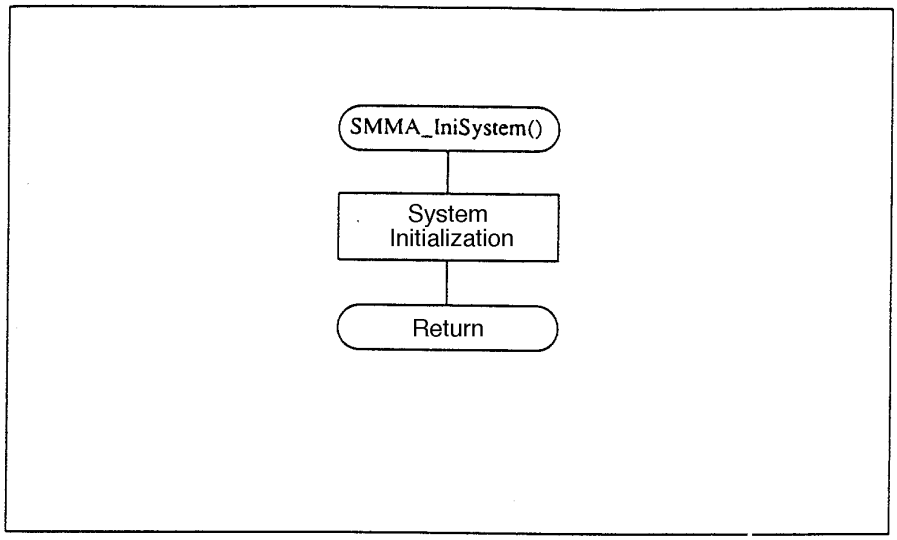


Figure 3 System Initialization Flow

SMLO_SegaLogo() Sega Logo Display

The Sega logo is displayed as shown in Figure 4. Because the game mode variables go to TITLE_MODE before returning, the title is displayed without further conditions.

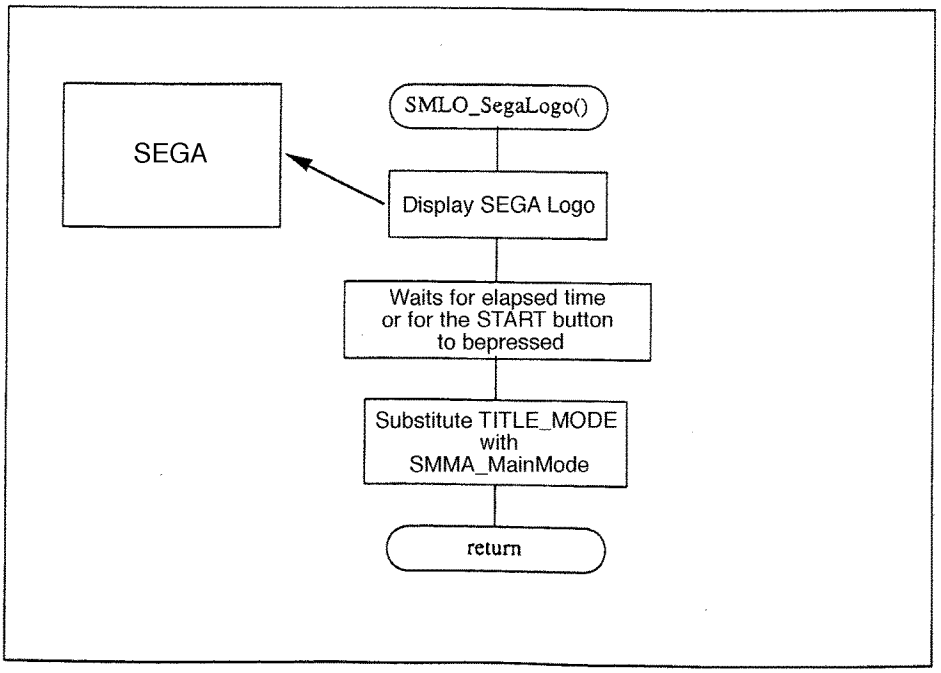


Figure 4 Sega Logo Display Flow

SMTI_Title() Title Display

The title is displayed as shown in Figure 5. After the START button is pressed, move to program select.

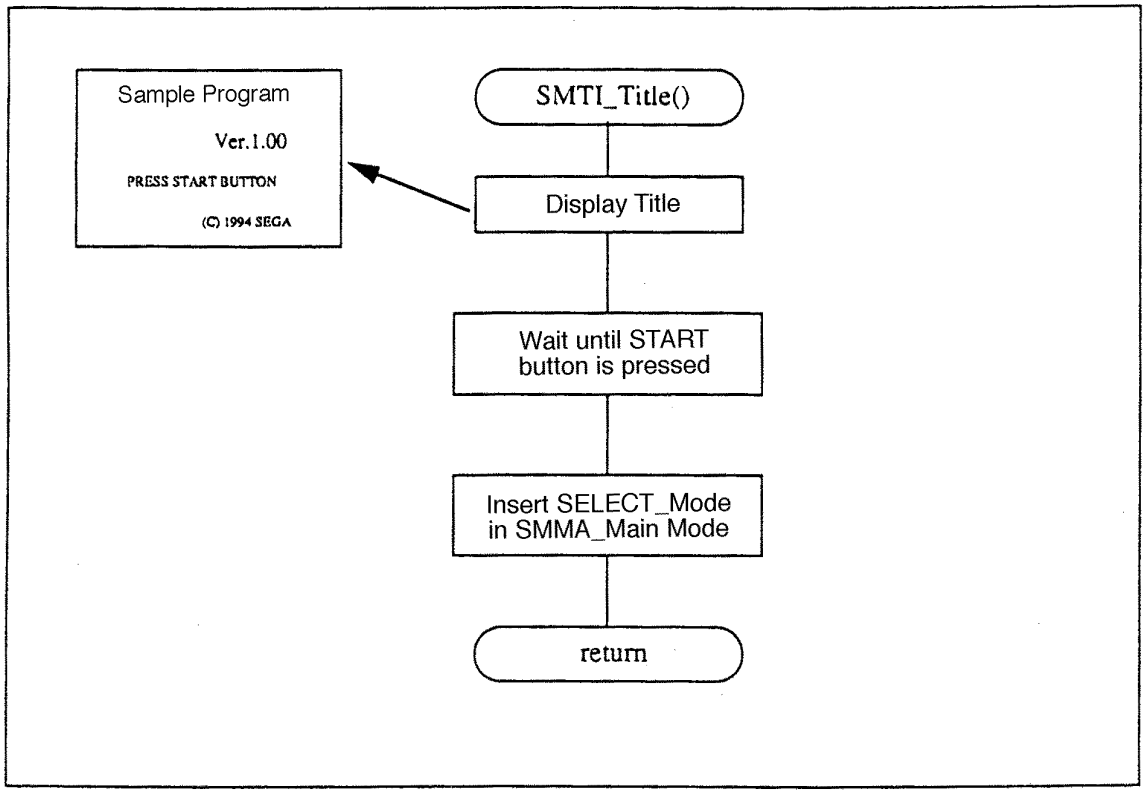
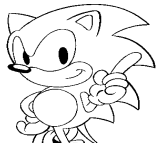


Figure 5 Title Display Flow



SMSL_ModeSel() Execute Program Select

The program select screen is displayed as shown in Figure 6. Use the D-pad to select, and C button to execute the sample program, game program, etc.

After the program ends, the game mode variables are set to SELECT_MODE before return so the Program Select is executed unconditionally. When EXIT is selected, the game mode variable is set to LOGO_MODE, and returned, so the transition occurs to the Sega Logo display unconditionally.

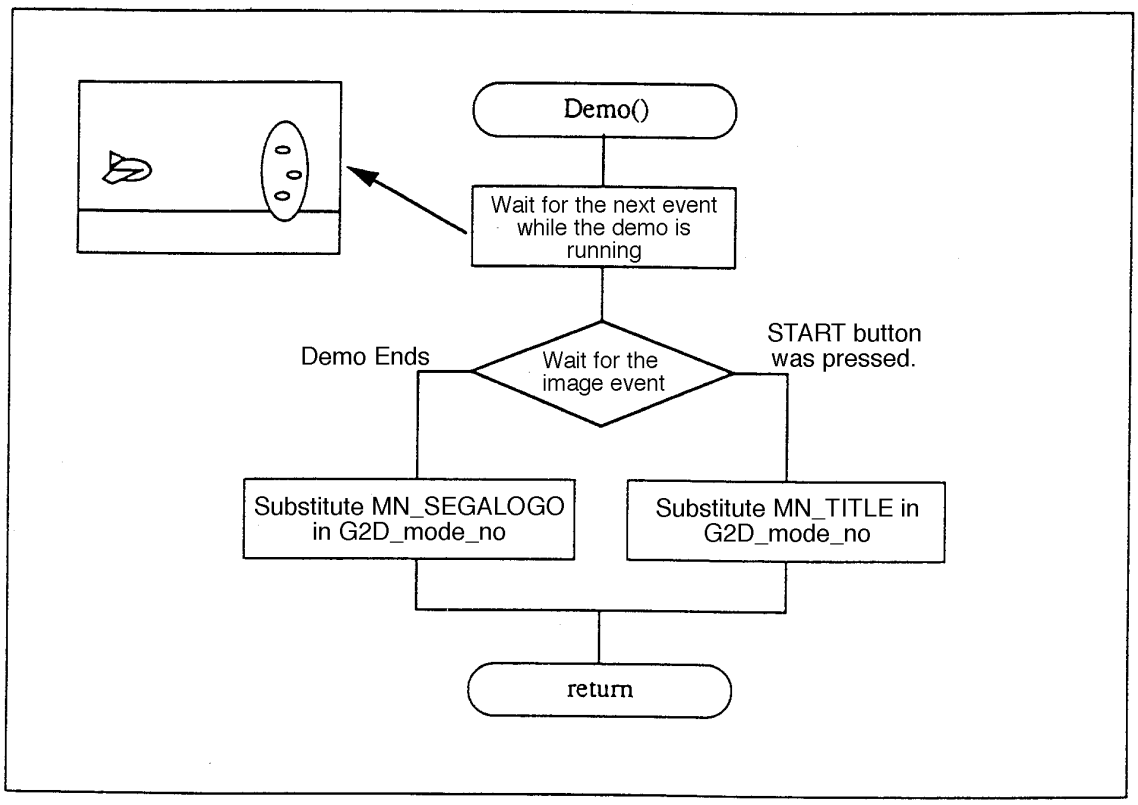


Figure 6 Demo Execution

The following items are in the Program Select Menu.

1. SCROLL SAMPLE
2. SPRITE SAMPLE
3. WINDOW SAMPLE
4. GAME SAMPLE
5. EASY SOUND TEST
6. <EXIT>

4. Basic Function Program

The basic function program is executed by selecting as scroll sample, sprite sample, window sample, or game sample from the game sequence program select. At first, a selection screen similar to the program selector screen appears. Use the D-pad to select, and the C button to execute.

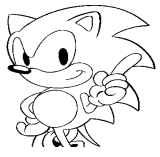
The scroll sample, sprite sample and window sample menus have the items shown below. Each item in these menus is configured to allow a simple program to be created.Refer to the source code for the contents of each program.

Scroll Sample

1. BITMAP SCROLL
   Draws points, lines and boxes. (Uses NBG0, bitmap format, RGB32768-color mode.)
2. NORMAL SCROLL
   Scrolls up, down, left, and right. (Uses NBG0, cell format, 256-color mode.)
3. LINE SCROLL
   Scrolls while shaking in vertical and horizontal directions. (Uses NBG0, cell format, 256-color mode.)
4. MULTI SCROLL
   Displays several layers simultaneously by giving a priority to the scroll. (Uses NBG0~3, each surface is cell format, 16-color mode.)
5. LINE COLOR SAMPLE
   Calculates the color for each line, lowering the translucency rate as it gets closer to the center of the screen.
6. ROTATE SCROLL
   Rotates scroll at an angle horizontally and vertically. (Uses RBG0, cell format, 256-color mode.)
7. <EXIT>

Sprite Sample

1. POLYGON TEST
2. POLYLINE TEST
3. TEXTURE TEST
   Rotates, enlarges and reduces polygons, polylines, and textures. (Changes the content of the sprite structure.)
4. HENKEI TEST
   Scales the peripheral of the sprite. (Defines 4 points and transforms the inside texture.)
5. LINE TEST
   Moves lines in the screen while changing colors. (Sets the position of 4 points and the move speed and varies the texture within.)
6. SHADOW TEST
   Displays the sprite or polygon as a shadow. (Displays a normal or MSB shadow while the sprite automatically calculates color while fading in and fading out.)



7. SPRITE TEST
   Rotates, enlarges and reduces the texture while moving it around.
8. SHADING TEST
   Adds different shades to each of 3 cubes. (Adds “no shading,” “flat shading” and “Gouraud shading.”)
9. 3D TEST
   Rotates, enlarges and reduces a polygon.
10. <EXIT>

Window Sample

1. NORMAL BOX WINDOW
   Displays a rectangular window.
2. NORMAL LINE WINDOW
   Displays a line window.
3. SPRITE WINDOW
   Displays a texture form window.
4. 3WINDOW SAMPLE
   Displays all windows.
5. <EXIT>

Game Sample

2D Shooting Game.

5. Action Control

What is Action

A method called *action* is used in the 2D shooting game included in the sample games to provide quasi-multitasking. Action is a function that processes the action and player, enemy, shots, and background scroll in 1 cycle.

This function is registered individually (makeaction) and the registered functions are run at the same time in a loop (actionloop) and executed to realize quasi-multitasking. This sample is supplied in smp_task.c and smp_tash.h and is used in most of the sample programs.

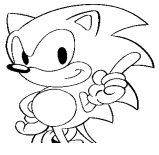
Action Control Makeup

Action control controls action from a single source using action structures. The basic action structure (ACTWK) is composed of the flag being used (id), level, mode, status, and function executable address (pcbuff) and an area that can be freely set.

Other than id and pcbuff, any of these can be used freely. Also, if additional information needs to be stored, it can then be added using a macro that is described later as a member. (See Figure 7.)

The 2D shooting game declares an exclusive action structure called GAMEACT. In order to manage several actions, the elements that make up ACTWK are defined in an array so that information can be saved for each action. This controls multiple actions. The maximum size of this array is set by ACTWKMAX. In other words, only a maximum number of ACTWKMAX can be registered.

There are two types of actions registered, those that are not used and have no conditions, and those that are registered in specified areas in the array. Actions that are registered are called up by the actionloop. The actionloop at this time passes the pointer to that action as an argument so that during the action, action information can be operated.



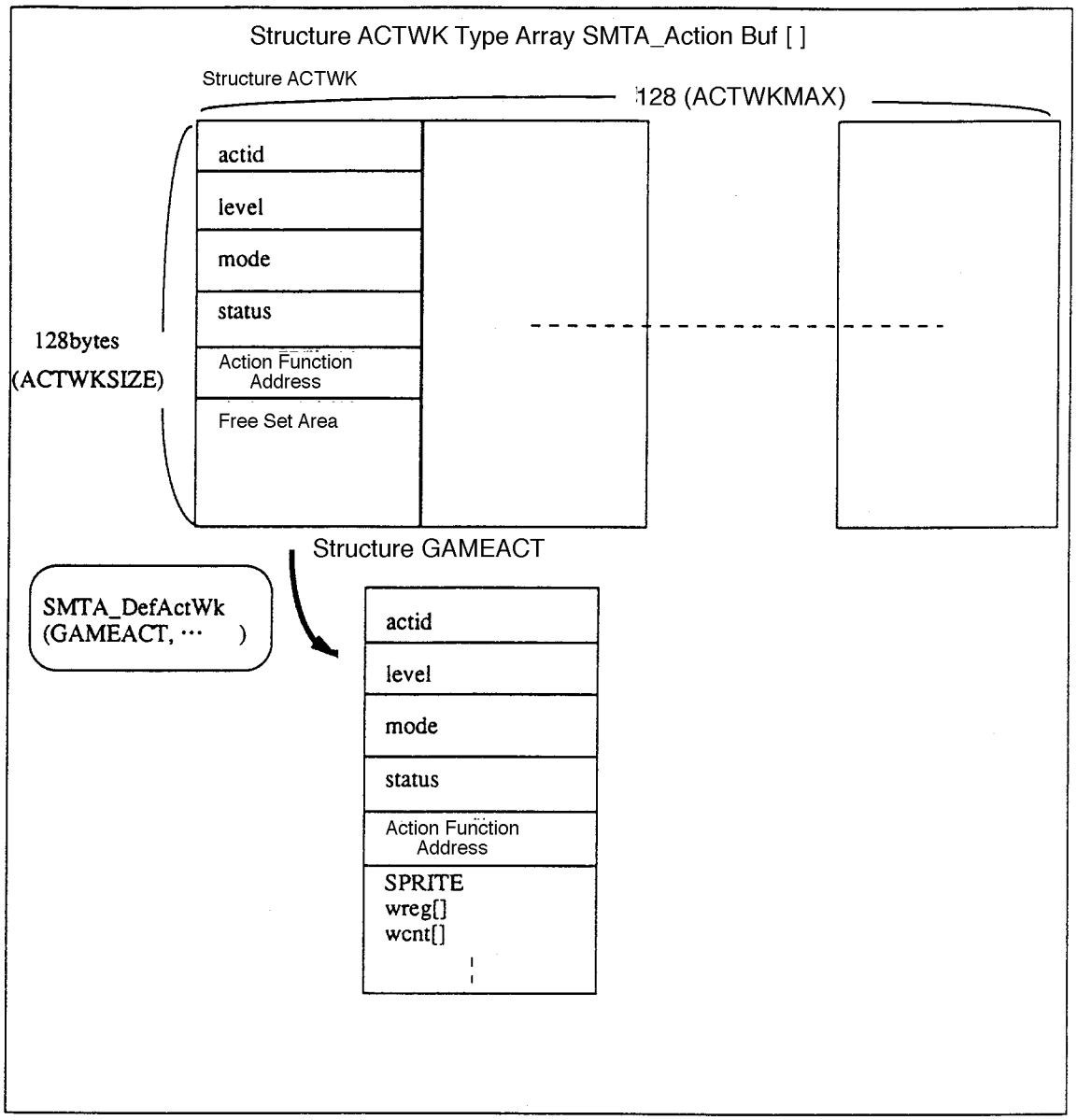


Figure 7 Basic Structure and Exclusive Structure

Explanation of theAction Control Function and Macro

The functions supplied in smp_task.c and smp_task.h are explained here.

[Function]
void *SMTA_MakeAction(void) *execadd)

This function searches for unused actions (id=0) and registers the executable address to the pcbuff. It then changes the action to active (id=1). In this function, the action number within the array is undefined. This causes problems when action information is compared. Shot and player collision (hit determination) is one example. If the order needs to be controlled, use the following function.

void *SMTA_MakeActionX(void *execadd, Uint8 start, Uint8 count)
Searches for unused actions from the start of the array for the count number, registering any unused actions along the way. In other words, the range is specified and the action is set. This function can be used in situations like shot actions where the same action is set in multiplicity. If the range in the array that holds the action is known, collisions can be easily secured.

void *SMTA_SetAction(void *execadd, Uint8 start)
Forces the action to be set in the start of the string. It is set whether or not the action is in use. It can be used in player action, and so on, that will only be set once.

void SMTA_ActionLoop(void)
Searches, in order, from the beginning of the array to the end, executing all of the active actions. The pointer to the action structure is passed to the action as an argument.

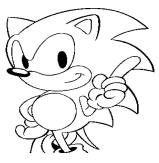
[Macro]
SMTA_DefActWk(actname, member)
Declares an exclusive action structure. This entails using a free set member. Refer to the program below for an example. This macro does nothing more than declare the structure by governing the action structure information uses.

SMTA_CheckAction(ACTWK *ptr)
This macro has a true (1) value when an action was registered and a false (0) when it did not. Refer to the example below for the usage.

[Example]
```
ACTWK *ptr;

ptr = SMTA_MakeActionX(ShotAct, 0, 8);
if (SMTA_CheckAction(ptr))
      /* Register OK */
else
      /* Register Failed */
```

SMTA_KillAction(ACTWK *ptr)
This macro changes the action to inactive by writing a 0 to the id, and normally is described during action, and by self set action to non-action. That action will not execute in the next actionloop.

[Example]
```
void PlayerAct(ACTWK *ix)
{
      if (Deadflag)           /* Player is killed */
         SMTA_KillAction(ix);
}
```

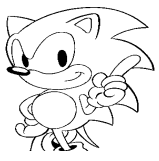



Example ofAction Control

Shows an actual example of use.

```
/*
 *     Variable definition
 */
Uint8    Mainlevel;
Uint8    Gameover;
SMTA_DefActWk(GAMEACT,
       SPRITE sprite;            /* SPRITE DISPLAY          */
       Sint16 wreg[4];           /* Universal use register  */
       Sint16 wcnt[4];           /* Universal use counter   */
       Uint8 colino;             /* collision table index   */
       Uint8  coliatr;           /* collision attribute     */
       Uint8 coliflg;            /* collision flag          */
       Uint8 atp;                /* kougeki ryoku           */
       Sint16 hp;                /* hit point               */
);            /* Declares exclusive action structure GAMEACT       */

/*
 *     Function Prototype
 */
void GameMain(void);
void PlayerAct(GAMEACT *);
void EnemyAct(GAMEACT *);
void ScrollAct(GAMEACT *);

/* ####[Action Example     Main Program ]#### */
void GameMain(void)
 {
   enum {
       INIT, MAIN,
   };

   Mainlevel = INIT;
   Gameover = 0;                 /* Game end flag           */
   for(;;) {
       intWait();                /* Wait for V blank        */
       switch(Mainlevel)   {
       case INIT;
             InitVdp();          /* Initialize VDP          */
             SMTA_ActWkInit()    /* Initialize action work  */
             SMTA_MakeAction(PlayerAct);
                                 /* Set player action       */
             SMTA_MakeAction(EnemyAct);
                                 /* Set enemy action        */
             SMTA_MakeAction(ScrollAct);
                                 /* Set background action   */
             Mainlevel++;
       case MAIN:
             SMTA_ActionLoop();
             if (Gameover) Mainlevel++;
             break;
       case EXIT;
             return;             /* Game end                */
       }
    }
 }
```

```
/* ####[ Player Action ] #### */
void PlayerAct(GAMEACT *ix)   /* (ACTWK *) >> (GAMEACT *) to cast *)
{
   enum {
       INIT, MAIN,
   };

   switch(ix->level) {
   case INIT:
       /* Initialize player coordinates, etc. */
       ix->level++;
   case MAIN:
       /* Move player             */
       /* Collision judgment      */
       break;
   }
}
```

Using V Blank

The game main processing structure is similar to the game sequencing structure. Included in that is the intWait function. The intWait function includes the screen V blank. There are two main reasons why the action loop is after V blank wait:

1) Reduces Flicker
   By executing the action after the V blank hold, the next object on the frame will have a new area to be displayed. If an object were to move between screen displays, the scanning line position and moving direction would cause poor display. Figure 8 shows this condition. (a) is displayed correctly because "movement" occurs during V blank, but (b) and (c) have movement during scanning causing two objects to be displayed at the same time, or nothing at all to be displayed.

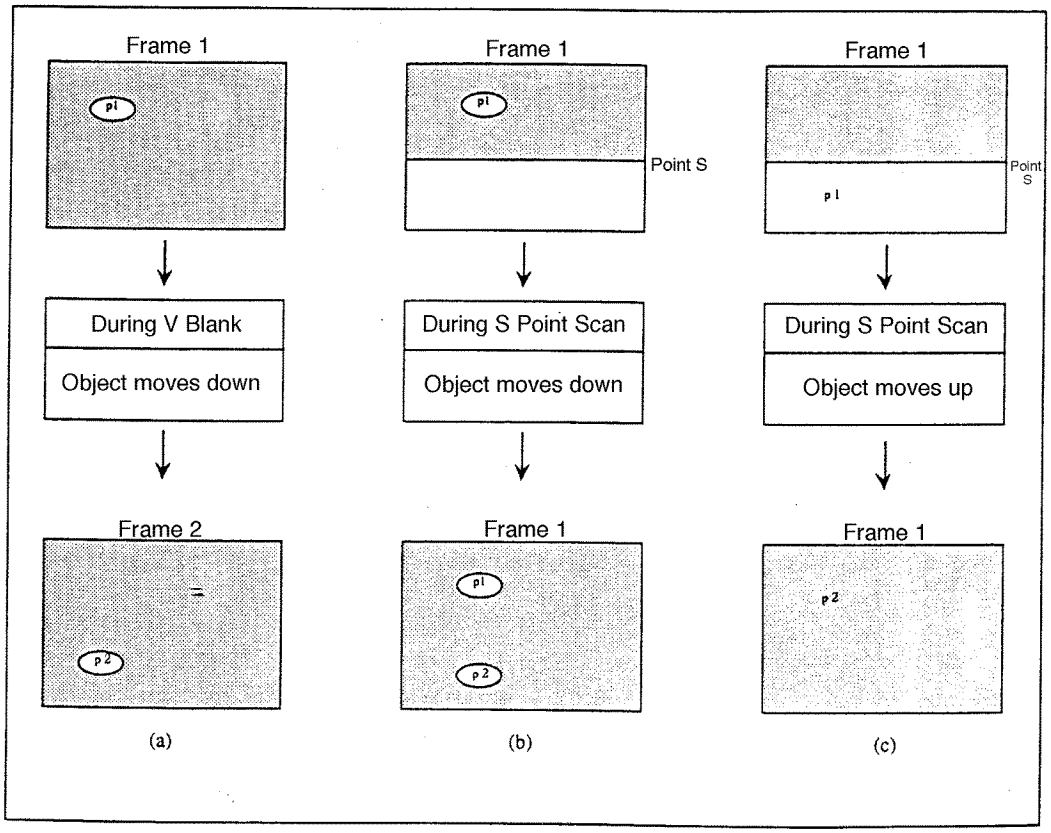


Figure 8 Relation of Scanning Lines and Objects

2) Timing
   Correct interval timing is required to get the character or background to move at a fixed (constant) speed. V blank can be handled at intervals of 1/60 of a second. Therefore, even if variation occurs by an amount equal to the processing of a single action loop, the motion speed can be kept constant (at a fixed speed). However, if the action loop cannot be completely processed during 1 frame, it cannot be used as an accurate timer.

6. Directory Structure, File Name, Function Name, Variable Name

Figure 9 shows directory structure.

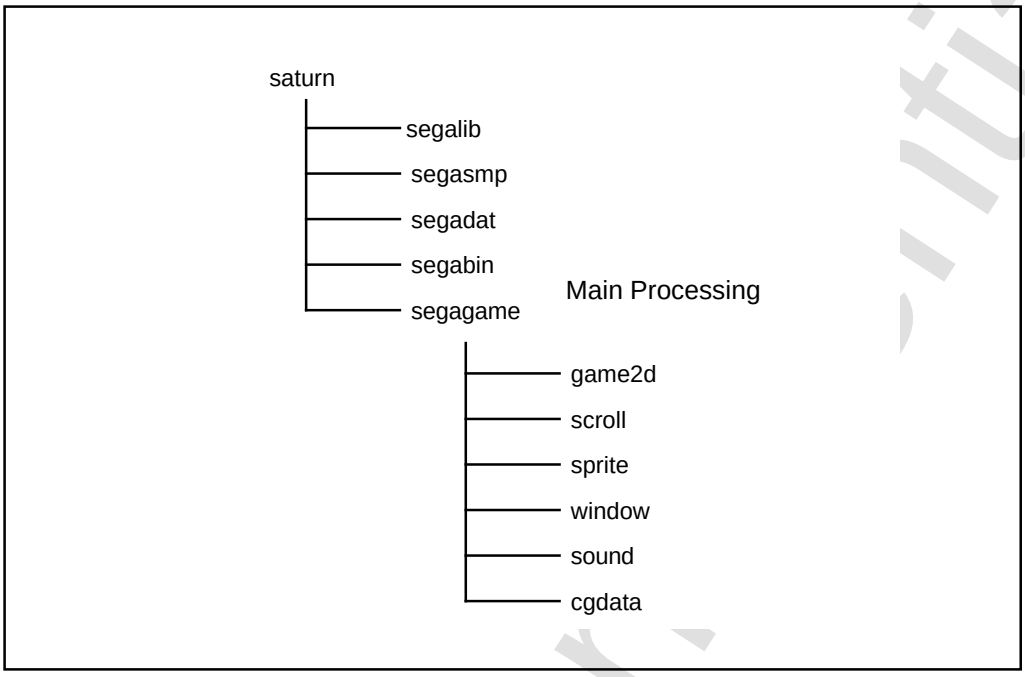
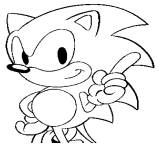


Figure 9 Directory Structure

All file names start with "smp_". Also, external variables and external functions start with "SM??_". See the table below for specific examples.

Filename	Function, Variable Prefix	Contents
smp_main.c	SMMA_	Main processing
smp_logo.c	SMLO_	Display logo
smp_slct.c	SMSL_	Display select
smp_titl.c	SMTI_	Display title
smp_task.c	SMTA_	Action control
scroll/smp_sc10.c	SMSC_	Scroll sample
sprite/smp_spr.c	SMSP_	Sprite sample
window/sm_wind.c	SMWI_	Window sample
sound/smp_snd.c	SMSN_	Sound sample



7. Compile/Execute Procedure

Compiling Procedure

PC Version

Execute **smpmk.bat** in the **saturn/segagame** directory.

```
C:/saturn/segagame>smpmk.bat [ENTER]
```

WS Version

Make smp.mk in the segagame directory.

```
saturn/segagame% make -f smp.mk  [ENTER]
```

Execute Procedure

PC Version

Execute the following commands from the debugger command line.

Command	Comment
rs [ENTER]	Reset the ICE.
g [ENTER]	Execute the BOOT ROM. If it won't boot up, reset the target.
l ;r:smp.abs [ENTER]	Load the sample program.
g 06010000 [ENTER]	Execute the sample program.

WS Version

Execute the following commands from the debugger command line.

Command	Comment
rs [ENTER]	Reset the ICE.
g [ENTER]	Execute the BOOT ROM. If it won't boot up, reset the target.
ftp WS name or IP address	[ENTER]
User name [ENTER]	
Password [ENTER]	
ll ;r:smp.abs [ENTER]	Loads the sample program.
g 06010000 [ENTER]	Executes the sample program.

8. SIMM/VCD Compatibility

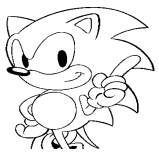
With normal software, data is not loaded (read) at the same time into work memory when it is executed, but is broken into modules which are read into memory each time it is required. The blocks that are resident in the work memory are only for initializing, reading, and executing each module. That resident block is called a kernel.

Actual game software is stored in a CD-ROM or cartridge ROM, and after (launching) reset, is in a form to be partitioned and loaded. This is emulated by SIMM or VCD. Using a sample program as an example, modularization will be explained.

An image of the work memory when the sample program is partitioned into modules is shown in Figure 10. Also, a listing of the kernels and modules in the sample are shown below.

```
Kernel
      smp_krnl.bin
Modules
      logo.bin
      titl.bin
      wind.bin
      scrl.bin
      gm2d.bin
      sprt.bin
      d214.bin
```

These programs are found in the segagam1 directory. Here, the rof2bin is used during "make" to convert the kernel and module abs format file to a bin format. Also, mfcat is used to convert each bin file to a SIMM file.



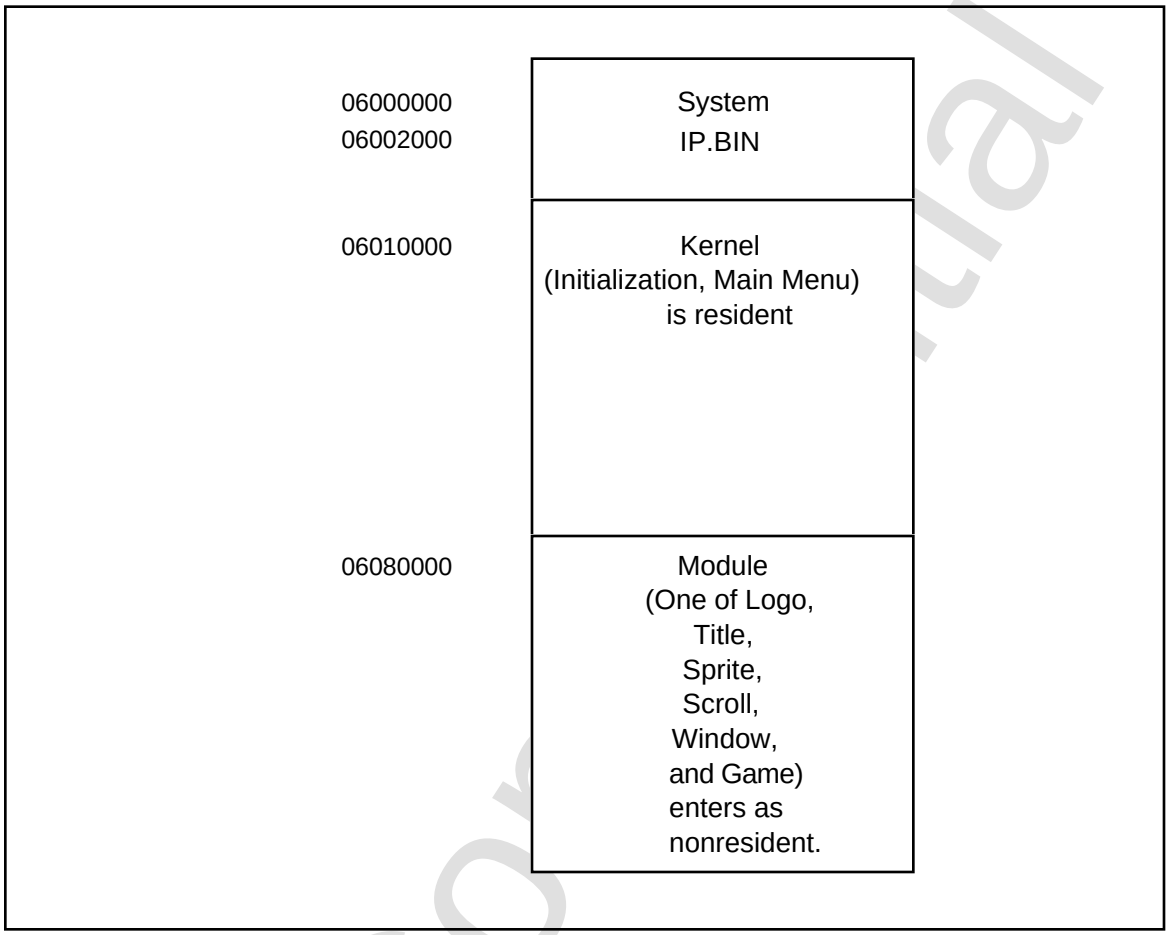


Figure 10 Sample Program Memory Image

Executing Procedure in SIMM File

1. Program Make
   WS Version make -f smp.mk[ENTER]
   PC Version smpmk.bat[ENTER]
   This will make a smp.lod that can be executed in a SIMM.

2. Program Loadin and Execution

```
1 04020000;m:smp.lod[ENTER]
  rs[ENTER]
  g [ENTER]
```

   After a brief display of the Sega logo, the sample game is executed.

VCD Executing Procedure

1. Program Make
   WS Version make -f smp.mk[ENTER]
   PC Version smpmk.bat[ENTER]

   Here, the program module *.bin is made. smplod is also made here, but is not required.

2. Building the Program CD-ROM Image and VCD Startup
   Create the CD image by copying the btsmpfs.scr and btsmpfs.pre located in the vcd directory, and the *.bin file which is a PC program for the VCD, and by executing the following commands in the vcd directory.
   `vcdpre btsmpfs.pre` [ENTER]
   `vcdbuild btsmpfs.pre` [ENTER]

   Start the VCD by moving the programming box switch to the VCD side and executing the following commands.
   `chev us` [ENTER]
   `vcdemu btsmpfs` [ENTER]

   Press any key at the first message and verify that SEEK2 is displayed in the command column.

3. Program Execution
   `rs` [ENTER]
   `g` [ENTER]

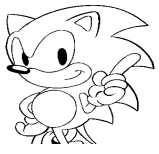
   The sample game will start after the Sega logo is momentarily displayed.

In both SIMM and VCD, rs and g start the boot ROM and execute read IP.BIN from SIMM or VCD. The sample is executed by reading the kernel from IP.BIN.

Procedure for the Module Partition

For partitioning to the kernel and module, symbol information is required by each side (mutually), and thus some steps are needed to be taken. To create the common information ABS file, use the method shown in Figure 11. The tool used to extract the symbols from the map file differ depending on the development environment being used, as shown below.

1. PC Version
   SYMADD.EXE Map File Name [ENTER]
2. WS Version
   awk -f kiri.awk Map File Name [ENTER]



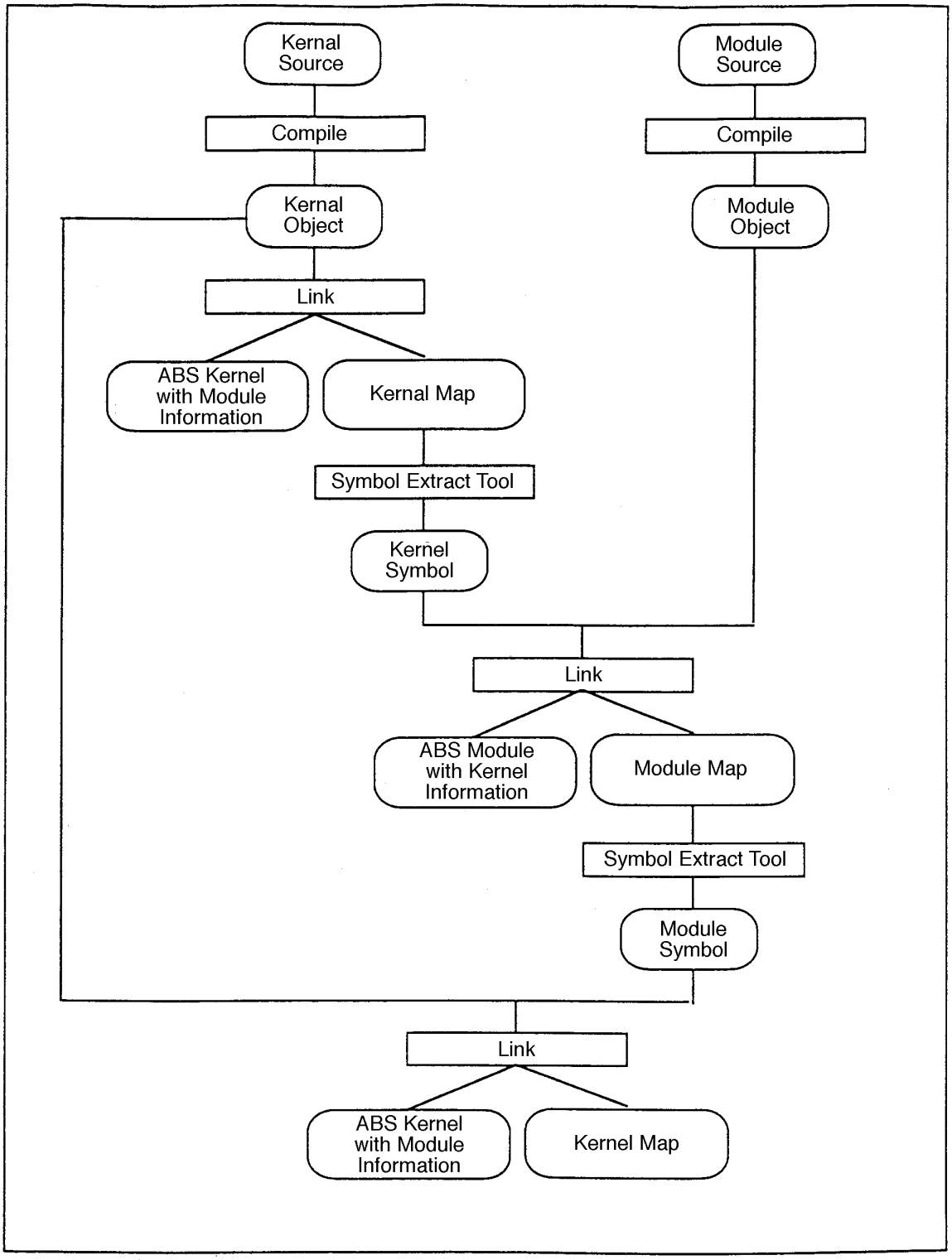


Figure 11 Module Partition Procedure

INDEX