



# Programmer's Guide

This guide describes how to use the SPSM in glass box mode. Glass box mode allows users to add new variables and parameters to the SPSM as well as the ability to alter the SPSM algorithms or implement new algorithms. The Microsoft C compiler is required for using the glass box mode.



Statistics  
Canada

Statistique  
Canada

Canada



## Table of Contents

Introduction.....	2
Purpose of the Glass box Mode .....	2
Hardware and Software Requirements for the Glass box Mode.....	4
Programming Knowledge Required .....	5
Operating System Knowledge Required.....	5
Fundamental Programming Concepts (Not Language Specific) .....	5
Knowledge of the C Programming Language .....	6
Quick Start Example .....	6
Preliminaries .....	7
Changing the project environment.....	8
Changing the Alternative Driver Function (Adrv . cpp).....	8
Changing the Alternative Family Allowances Function (Afamod . cpp).....	9
Testing the Resulting Alternative SPSM Model.....	11
Summary .....	14
SPSD/M and Glass Box Directory Structure .....	14
The SPSD Household/Individual Structure .....	16
The SPSD/M's Data Structure.....	17
Introduction to Pointers in the SPSD/M .....	18
The Bestiary .....	18
Examples of Looping:.....	19
References With Respect to an Individual:.....	21
Summary .....	22
SPSM Function Calling Structure.....	22
Glass Box Development: Adding Typical Scalar Parameters .....	23
General Procedure for Making Glass box Changes: A Recapitulation.....	24
Create Task Sub-directory .....	24
Identify Files to be Changed .....	24
Copy Relevant Files to Task Sub-directory .....	24
Edit Those Relevant Files .....	25
Compile the new version .....	25
Test the New Version of the Model.....	25
Carry Out the Intended Analysis.....	25
Introduction to Parameter Addition .....	25
Copy Files Adrv . cpp , Mpu . h , Ampd . cpp , Afamod . cpp ,	
SPSMGL . dsw.....	26
Update the project.....	27
Update the Algorithm Description in Adrv . c .....	27
Modify Mpu . h to Define the new Parameters.....	28
Modify Ampd . cpp to Make the Parameters Available to the SPSM .....	28
Modify the Functions that Use the New Parameter(s).....	30
Validate and Make Black-Box Production Runs .....	31
Summary/Conclusion.....	32
Glass Box Development: Adding Less Typical Parameters .....	33
pmaddent: The Function and its Arguments.....	33

Characterizing Scalar Parameters .....	37
REAL/float/NUMBER Parameters .....	37
INTEGER/int Parameters .....	37
FLAG Parameters .....	37
FRACTION Parameters .....	37
OPTION Parameters .....	38
EDIT-FRACTION Parameters .....	38
DUMMY Parameters .....	38
Vectors of User-defined Parameters .....	38
Additions to Mpu.h, Cpu.h or Apu.h .....	39
Additions to Ampd.cpp .....	40
User-Defined Parameter Vector References in the Source Code .....	41
Specification of Parameter Vector Values .....	41
Summary .....	42
User-defined Schedules for Lookups .....	42
Schedule Types and Lookup Functions .....	43
Appearance in SPSM Header Files .....	43
Appearance in pmaddent Calls in Ampd.c .....	44
Employing Schedule References in User Code .....	45
Appearance in Parameter Files .....	46
Key Points for Adding Schedule Parameters .....	47
Adding Matrices of Parameters .....	47
Appearance in Mpu.h .....	48
Appearance in Ampd.c .....	48
Referencing Matrix Elements in Source Code .....	49
Appearance in Parameter Files .....	49
Summary/Conclusion .....	50
Glass Box Development: Adding New Variables .....	51
Overview for Adding Variables .....	51
Dependent Variable Types and Characteristics .....	52
The vardef and stradd Functions and their Arguments .....	52
Vardef "Name" Argument (and Definition of Variable "Stem" Name): .....	53
Vardef "Home Structure" Argument: .....	53
Vardef "Variable Location" Argument: .....	53
Vardef "C-Type" Argument (C_NUM & C_INT): .....	54
Vardef "Usage" (Type) Argument (V_ANAL & V_CLAS): .....	54
Stradd Calls for Numeric Analysis Variables: .....	54
Stradd Calls for Integer Analysis Variables: .....	55
Stradd Calls for Integer Classification Variables: .....	55
The Family Allowance Supplement Example Extended .....	56
Changes to project files and Adrv.cpp .....	57
Changes to vsu.h .....	57
Changes to vsdu.c .....	58
Changes to Afamod.cpp (Or, more generally, any new substantive source code) .....	58
Identifying String .....	59

Local Variables .....	59
Calculate and Assign the New Model Variables .....	59
Compilation.....	61
Validation.....	61
Summary/Conclusions .....	63
Changing Base and Variant Data Variables.....	64
Making Changes That Affect All Tax/Transfer Systems in a Model: .....	65
Typical Income and Population Growth Changes Via APR/API Files .....	66
Changes Involving New Logic For <code>adju.cpp</code> .....	66
Adding New Database Adjustment Parameters.....	67
A Worked Example.....	68
Checklist for Changing Database Variables "Globally" .....	72
Making Changes That Affect Only the Base or Only the Variant.....	72
Implementing Changes in <code>Acall.cpp</code> .....	74
A Worked Example.....	75
Checklist for System-Specific Database Changes .....	82

## Introduction

The *Programmer's Guide* describes how users can alter the SPSM in order to model tax/transfer systems or policy options not directly addressable by the SPSD/M as distributed; e.g. they might make an alteration to the logic of the tax/transfer system in order to assess the static distributional impacts that would result from a policy proposal.

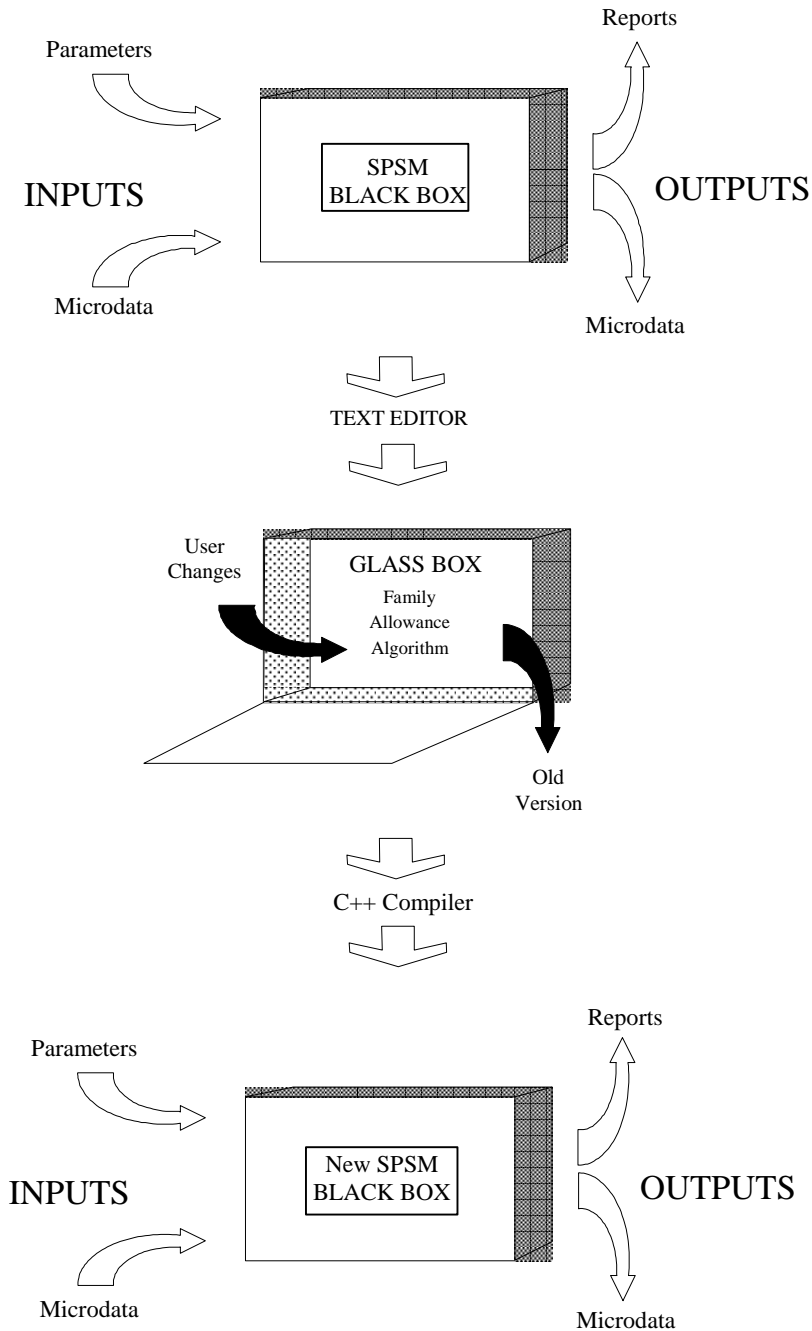
This chapter introduces a variety of preliminary topics critical for understanding the use of the SPSM in its glass box mode. Specific chapter topics include:

- (1) a description of the glass box mode, especially in contrast to the black box mode,
- (2) the hardware and software requirements for using the glass box mode,
- (3) the degree of programming knowledge required.

Subsequent sections in the guide then take up the details of actually developing glass box applications. Thus, the following section describes a "*Quick Start*" procedure that tests the success of the SPSM installation by effecting a simple glass box modification to the SPSM as distributed. The Section entitled *SPSD/M and Glass Box Directory Structure* characterizes the subdirectory structure relevant to the various aspects of glass box operations. The *SPSD Household/Individual Structure* provides critical details on the key SPSD data structures used by the SPSM. *SPSM Calling Structure* describes the calling structure of the SPSM modules that make up a specific model. *Glass Box Development: Adding Typical Scalar Parameters* addresses the mechanism for adding user-defined model parameters to an SPSM model, treating the most common forms of scalar parameters. *Glass Box Development: Adding Less Typical Parameters* then takes up the addition of less typical kinds of scalar parameters, as well as the addition of vectors and matrices of new model parameters. *Glass Box Development: Adding New Variables* describes the addition of new variables to a model. *Changing Base and Variant Data Variables* provides the definitive statement on managing standard and alternate algorithms within the context of glass box operations.

### **PURPOSE OF THE GLASS BOX MODE**

A simplified systems view of the process of simulating taxes and transfers may be as follows:



A user specifies a series of inputs (parameters and data) which are then processed through a system of algorithms (the Black Box) which in turn produces a system outputs (tables and microdata). The user may create many different simulations by varying the inputs and then analyzing the outputs. He may even deduce some of the contents of the black box through repeated testing. However, the simulations possible are limited by the contents of the black box. If, for example, the rules of the Manufacturer's Sales Tax are not included in the system of algorithms (with provisions for appropriate input data and parameters) then this program can not be simulated without actually opening up and changing the black box. This ability to look inside the black box and alter its contents is like turning the black box into a glass box.

This guide explains how to use the SPSM in its glass box mode. Specifically, the term "glass box mode" refers to a method of modifying versions of the executable SPSM program to accomplish analyses that are not possible with the original, unmodified SPSM. Glass box mode may be used to add or modify parameters, variables, and algorithms. Using the "glass box" mode always entails altering the C++ Language source code and recompiling an executable version of the program. The "black box mode" refers to the subsequent execution of an executable version, either as shipped by Statistics Canada or as modified by user's in "glass box" mode. It is always through the black box mode that a user carries out a variety of policy relevant simulations via parameter changes, user variables, and tabulation expressions.

Because of the extra steps involved, users should attempt to avoid the glass box mode wherever possible. The SPSM provides a number of devices that enable analysts to achieve many desired results without re-programming. The most common technique is to alter the default sets of program parameters that drive the SPSM. The analyst could simulate the impact of an increase or abolition of Family Allowances by changing the numeric values of the relevant parameters. In a second example, the analyst can define their own variables in the control parameter file, and can use the resulting variables in a whole range of SPSM outputs. The *Introductory Guide* provides an extensive detailed example in which an analyst uses the user-defined variables to simulate an earned income tax credit. Similarly, the analyst can create variables "on-the-fly" as expressions and export or tabulate them just as if they had been full-fledged variables, and can conveniently represent differences between a given variable in the base and variant tax/transfer systems. The *XTab User's Guide* provides several examples of this type of on-the-fly definition.

The Glass box mode must be used under the following conditions:

- (1) Adding any new parameters.
- (2) Adding new variables that require reference to other specific family members.
- (3) New proposals which are designed to interact with the tax/transfer system. For example, a taxable newborn allowance.
- (4) New proposals that alter the logic of existing programs in ways that have not yet been parameterized.

When users need to make such changes in the SPSM to reflect alternative tax/transfers systems, they need to be familiar with the techniques described in this guide.

## **HARDWARE AND SOFTWARE REQUIREMENTS FOR THE GLASS BOX MODE**

The *Installation Guide* provides the definitive statement on hardware and software requirements. For most users, a printer is a practical necessity. The discussion here assumes that one is present.

The key aspects of software requirements are as follows:

1. Use of the SPSD/M in the glass box mode requires the availability of Visual C++ that serves to compile the user's C language source code statements into the machine language format required by the SPSM.
2. The SPSM itself requires an operating system compatible with the version of Visual C++.



3. The user must have an appropriate editor for entering or altering C language source code, such editing being central to glass box usage
4. It is suggested to use **an efficient text editor compatible with C++ code.**

Users expecting to use the SPSM heavily in the glass box mode will probably also want the added efficiency provided by "utility" software such as the MKS toolkit that makes many Unix style features available within the operating system.

## **PROGRAMMING KNOWLEDGE REQUIRED**

Because use of the SPSM in the glass box mode requires the user to do some programming, glass box user will have to be somewhat more knowledgeable than the typical black box user. This section characterizes the kinds of things that a glass box user will either have to know or be prepared to learn.

### **Operating System Knowledge Required**

Using the SPSM in its glass box mode requires that the user be fairly comfortable with a number of areas relating to the operating system. A user needs to know about disk drives, files, and file naming conventions, and about directories and subdirectories.

The user should be familiar with the concept of the DOS environment and with environment variables such as the `PATH` variable. Effective operation in the SPSM's glass box mode also requires that users be proficient with a number of DOS commands. The DOS commands most critical include:

<code>DIR</code>	List directory contents
<code>TYPE</code>	List file contents
<code>MKDIR</code>	Make new directory
<code>CHDIR</code>	Change current directory
<code>RMDIR</code>	Remove directory
<code>COPY</code>	Copy files
<code>XCOPY</code>	Copy files and/or directories
<code>DEL</code>	Delete file
<code>SET</code>	Set/display environment variables
<code>PATH</code>	Display current path

Users who are not at ease with the concepts and commands described here will probably avoid a great deal of frustration by spending some time with the DOS manual or develop some ability to do it in Windows environment before tackling actual glass box applications.

### **Fundamental Programming Concepts (Not Language Specific)**

The SPSM glass box is not the place to learn your first programming language. Users should be familiar with at least one high-level computer language prior to using the Glass-Box (e.g. FORTRAN, BASIC, PASCAL, and SAS). Because glass box applications involve programming in a compiled language, it is desirable that glass box users come to the task already familiar with the key concepts. A user should be comfortable with the idea of using a text editor to write or revise source code, and with the idea of using a compiler to produce the

desired executable file. The user will benefit from a familiarity with the notions of libraries, macros, modular programming and program validation.

More critically, a user's experience with these concepts should be applied. Preferably, before tackling SPSM glass box applications, a user should already have written and debugged several non-trivial computer programs, not necessarily using the C language. Although it may be possible for a user to learn to program by using the SPSM, we recommend against the attempt. For prospective SPSM users needing to build or reinforce basic programming skills, a wide variety of programming texts are available.

### **Knowledge of the C Programming Language**

Because SPSM glass box applications involve programming in the C language, a user must also program in C. Although the structure of the SPSM means that certain things like input/output are done for the user, the prospective user will be most efficient if the basics are previously understood. Users have to understand the purpose of defining constants and declaring variables, and must appreciate the scopes of these declarations. They must understand variables and variable types, specifically including pointer variables and structured variables, and how the C language uses them. They must understand the nature and structure of functions and the variety of statements that comprise them. They must be familiar with C's major flow of control statements (if-else, switch, while, for, do-while), as well as C's table of assignments and operators, including the increment operator. For users who have worked in other programming languages and are capable of absorbing this information in a concentrated form, Kernighan and Ritchie's book, "The C Programming Language" is the standard reference. Similarly, the C language manual that comes as part of the Microsoft C Optimizing Compiler is a very useful and authoritative source for information about C and its implementation.

Finally, of course, SPSM users must understand the basics of the Microsoft C Compiler. It is also necessary to understand the thrust of what is going on, and the various error messages that the compiler may give in response to the user's code. The authority on these topics is, of course, Microsoft's set of manuals for the C compiler.

### **Quick Start Example**

As its title suggests, this chapter provides the user with a quick start at using the SPSM in its glass box mode. The chapter serves three main functions. First, it allows the user to check the installation of the compiler and SPSD/M. If the user can carry out the chapter's simple example successfully, then all of the major portions of the installations were performed properly. Second, the example introduces key glass box concepts and terminology. Third, the example illustrates, in an integrated manner, the general flow of glass box applications.

The chapter's approach is primarily narrative. Taking the reader through all the steps of a simplified glass box application, it concentrates on the general approach. It describes the key details of the exercise, but does not attempt to be exhaustive. The particular illustration used here was selected for its simplicity; it addresses the most critical aspects of glass box applications, but doesn't get bogged down in the additional requirements associated with more ambitious applications.

Substantively, the example models a relatively simple change to a single transfer program, Family Allowances, in the tax/transfer system. Our hypothetical analyst, intrigued by the practice of Prince Edward Island in the 1970's, seeks to ascertain the aggregate and distributional impacts that would be associated with giving additional Family Allowance benefits to larger families. More specifically, in the variant system, the analyst wants to increase the amount of the federal Family Allowance by \$10 per month per child for selected children in selected families. When a family has three or more children currently aged 0 through 17 years of age, then it receives, over the year, an additional amount equal to \$120 times the number of these "excess" children, i.e. \$120 for a three child family, \$240 for a four child family, etc. We assume that this additional Family Allowance benefit would be paid by the federal government to the usual recipient and that the benefit would be treated just like the regular federal Family Allowance benefit.

As regards the narrative, readers should not worry about the "whys" of the implementation. Subsequent sections in this *Programmer's Guide* will address all of them more fully. However, it is highly desirable that the user work through the example to the point of actually carrying out all of the tasks described. Only in this way can the first purpose, confirmation of the installation processes, be realized.

## PRELIMINARIES

The user should begin by selecting a subdirectory in which to work. This is the hard disk subdirectory in which the user will edit copies of the relevant C++ language source code files and describe the nature of the alternative system. **We strongly recommend that the user employ a directory other than those that the SPSD/M installation establishes for the SPSD/M itself.** The user can make a new subdirectory if necessary. For purposes of this narrative, we'll assume that a subdirectory named GLASSEX1 is available as the working subdirectory.

The user begins the process by copying, from the SPSD/M's GLASS subdirectory, to the GLASSEX1 work subdirectory, all of the relevant template files. Template files are files that already contain most of the necessary information for a glass box application, and which the user will modify to create the final versions necessary for the application. For this example, the relevant template files are as follows:

1. `Adrv.cpp`, the alternate "driver" template that invokes all of the SPSM's tax/transfer functions in the correct order. This template, distributed as part of the SPSM, is effectively a duplicate of the base driver function (**the user should copy it in its working subdirectory**).
2. `Afamod.cpp`, the alternate Family Allowances template that effects the computation of the Family Allowance benefit. This template, distributed as part of the SPSM, is effectively a duplicate of the base system's `famod.cpp` function that computes Family Allowance benefits. (**the user should copy it in its working subdirectory**).
3. `SPSMGL.dsw` and `SPSMGL.dsp` carry out the compilation and linking of the user's new model (**copy these files from /pspm/glass to your working subdirectory**).

For other glass box applications the user may also need to copy other tax/transfer templates and/or C language header files. In this example, however, the user does not need to alter any of the header files because the new model creates no new variables and uses no new formal parameters.

The general procedure for our illustrative glass box application is straightforward.

1. Working on COPIES of `Adrv.cpp`, `Afamod.cpp`, and `SPSMGL.dsw`, we make the small number of changes as described below.
2. Then we invoke/execute the `SPSMGL.dsw` utility in C++ to generate a working space. To work with the new model, the project should be recompiled to produce a new executable file (We assume the user knows how to proceed).

## CHANGING THE PROJECT ENVIRONMENT

Project environment should be modified if the user wants to change the name of the compiled `SPSM.exe` associated with the project in **Project: Setting: Link** to `GLASSEX1.EXE`.

The new files `Adrv.cpp` and `Afamod.cpp` must be included in the project (**Project: Add to project: Files**).

The key subdirectory `\SPSM\DEFS` should all be added in **Tools: Options: Directory**, since definitions relevant to glass box applications reside there.

## CHANGING THE ALTERNATIVE DRIVER FUNCTION (`ADRV.CPP`)

`Adrv.cpp` contains two kinds of information that the glass box user will want to alter. The first kind consists of labeling information that the SPSM uses in its reports and error messages. When the user makes appropriate changes here, the resulting output becomes more informative. The second kind consists of the function calls that effect the substance of the model's tax/transfer calculations.

The user makes the labeling changes in the portion of the code, starting at about line 50, that looks as follows:

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "Unnamed";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Untitled"
```

The **`ALTNAME[IDSIZE+1]`** string provides an identifying name for the alternative driver; the user replaces the placeholder "Unnamed" with the more informative name "FA Quick Start". The new name must not exceed 20 characters in length. This alternative name will then appear in the greeting screen. The **`Tdrv[]`** string provides a title for the alternative driver; the user replaces the placeholder "Untitled" with the more informative title "FA Quick Start". The new title may not exceed 20 characters in length. `TDRV's` contents appear as information in the control parameter file as an algorithm description. Upon completion of these substitutions, the revised "labeling section" appears as follows:

```
/* ===== GLOBAL VARIABLE DEFINITIONS ===== */
```

```

/*global*/ char ALTNAME[IDSIZE+1] = "FA Quick Start";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "FA Quick Start"

```

**In the substantive portion of the code, the user needs to make only a single change to indicate that the calculation of benefits for the variant system should use an alternate Family Allowance calculation.**

The relevant portion of the code, a single line appearing at about line 125, appears as follows:

```
famod(hh); /* compute family allowances */
```

Unmodified, it invokes the regular Family Allowance calculation. The user changes the line to invoke, instead, the alternative Family Allowance calculation that we shall describe shortly. The modification consists solely in the substitution of the new function name, and the revised source code appears as follows:

```
Afamod(hh); /* compute family allowances */
```

For this quick start example, these three simple changes constitute the entire set of modifications for the `Adrv.cpp` function.

### **CHANGING THE ALTERNATIVE FAMILY ALLOWANCES FUNCTION (AFAMOD.CPP)**

The `Afamod.cpp` function carries out the calculation of Family Allowances for the alternative system. In a manner analogous to the `Adrv.cpp` changes, the user's changes fall into two categories, labeling changes and substantive changes.

The labeling change is very straightforward. At about line 54, the function provides for a title, `Tfa[]`, for the module, with the title being used in the report in which the SPSM indicates the functions used to calculate the taxes and transfers. As with the title for the driver, this title appears as an algorithm description in the control parameter file. The relevant portion of the code appears as follows:

```

===== GLOBAL VARIABLE DEFINITIONS ===== */
/* Give global string describing version of this module */
/*global*/ char FAR Tfa[] = "Untitled"

```

The user changes the "Untitled" string to something rather more informative. The resulting section then appears as follows:

```

===== GLOBAL VARIABLE DEFINITIONS ===== */
/* Give global string describing version of this module */
/*global*/ char FAR Tfa[] = "FA Quick Start"

```

The substantive portion of the `Afamod.cpp` changes is a bit more complicated, but not extremely so. The option to be examined affects directly three of the calculated variables,

1. taxable Family Allowances (tfa),
2. federal Family Allowances, (ffa) and
3. Family Allowances, (fa).

(Of course other variables in the model, e.g. calculated taxes, are also affected indirectly.) When the number of children in the census family (the variable "nch") is three or more, we wish to increment each of the three Family Allowance variables by \$120 times the number of "excess" children. Everything else relating to the impacts of this policy change, e.g. the tax

impacts, will be taken care of automatically by other portions of the SPSM. In any event, the variables in the routine are temporary, ceasing to exist once execution leaves the Afamod function; only items that have been saved into the relevant portions of the household structure will be able to affect calculations elsewhere in the system.

With the nature of the desired change clear, the major remaining issue is where in the Afamod.cpp function to make the change. For purposes of logical correctness and clarity, the change should be made after the three variables have already had assigned to them the "base system" amounts of Family Allowances, but before any calculations such as assigning the amounts into variables in the data structure for the household. In this example, the changes can all be made, in parallel, at the same location.

### **The example is not valid anymore and will be revised**

The critical portion of the source code, as it exists before the implementation of our changes, appears as follows: (The DEBUG statements shown here are irrelevant to the normal calculation of Family Allowance benefits. Their presence permits detailed tracing to be performed when needed, but is irrelevant here except as it identifies the portion of Afamod.cpp, about line 366, where the Family Allowance changes will go.)

```
else {
    DEBUG1("%s standard FA calculation\n");
    tfa = nch * MP.STDFA;          /* taxable family allowances */
    ffa = tfa;                    /* federal part of family allowances */
}

DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

Substantively, we wish to add the expression "(nch-2) \* 120.0" to each of the three key variables, taxable Family Allowances (tfa), federal Family Allowances (ffa), and Family Allowances (fa). Further, such increments are appropriate only when the number of children aged 0 through 17 in the census family is at least three. C's "if" statement and its "+=" operator provide a very convenient way to do this.

```
else {
    DEBUG1("%s standard FA calculation\n");
    tfa = nch * MP.STDFA;          /* taxable family allowances */
    ffa = tfa;                    /* federal part of family allowances */
    /* $120/yr bonus for 3rd and subsequent children <18 */
    if (nch >= 3) {
        tfa += (nch-2) * 120.0;
        ffa += (nch-2) * 120.0;
    }
}

DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

With the completion of the changes to Afamod.cpp, the user's real work in implementing the changes is now essentially done. All of the relevant substance and labeling changes are complete and, assuming there have been no errors during their entry, all that remains is the compilation of the new model and then its validation. Most important, though, it is the resulting executable file from C++ compile, in this example GLASSEX1.EXE, that the user runs to analyze the impacts of the change that was modeled.

## TESTING THE RESULTING ALTERNATIVE SPSM MODEL

With all of the changes made, and the resulting files compiled and linked to create the new executable file, we are ready to test the new model. The two related goals of this step are:

1. to seek evidence about whether we have successfully made the desired change, and
2. to generate outputs that will help us diagnose errors should we have made any.

A very natural form of evidence takes the form of crosstabulations from a comparative run that uses the unmodified tax/transfer system as its base system and the modified form as its variant system. Later in this section we offer examples of two such crosstabulations.

In order to make the desired comparative run of the new model and get the output we need, we must alter the control parameters for the model. The *Parameter Guide* provides the authoritative description of SPSM control parameters; here we simply list the key parameter values for our purposes: (The "glassxla" portion of the two file names is an acronym for "Glass box example 1, version a".)

```
OUTCPR glassxla.cpr # Name of control parameter file (out)
VARALG FA Quick Start # Name of variant algorithm
VARMETH 3 # Method of creating variant variables
BASMETH 2 # Method of creating base variables
OUTTBL glassxla.tbl # Name of report file (out)
```

Two tables will suffice for validation in this example:

1. tabulate number of census families, variant federal Family Allowances, base federal Family Allowances, and their difference, all by number of children aged 0-17 (to show that we are giving the new Family Allowances to the right units in the right amounts) and
2. tabulate "delta Family Allowances" and "delta disposable income" by census family type to show both that we are giving the new FA to only the right kinds of units and that a part of it is being recovered via the tax system, with the recovery fraction higher for two-parent families than for one-parent families.

The XTSPEC parameter to generate these tables will look as follows:

```
XTSPEC
CF: cfnkids+ *
    {units,
    _imffa: L="Base Family Allowance (M)",
    imffa: L="New Family Allowance (M)",
    imffa-_imffa: L="Family Allowance Increase (M)",
    (imffa-_imffa)/units: L="Average Family Allowance Increase"};
CF: cftype+ *
    {imffa-_imffa: L="Family Allowance Increase (M)",
    immdisp-_immdisp: L="Disposable Income Increase (M)",
    (immdisp-_immdisp)/units: L="Mean Disposable Income Increase"};
```

The highlights of this request are as follows:

1. The first table uses "cfnkids" (number of children 0-17) as the row control variable. Note that cfnkids is an SPSD classificatory variable, while the variable "nch" used above to

effect the changes inside Afamod.cpp is a local variable that is defined as a "float" variable and could not be used here for tabulation purposes, even if it were classificatory.

2. The tabulated variables used in the first table are precisely those described above, numbers of families, new and old Family Allowance benefits and their difference.
3. The second table simply tabulates, for another existing classificatory variable, the differences in Family Allowances and in disposable income, with the "underscored" variables referring to the base system and the non-underscored variable names to the variant system.

The tables that result when one executes the new GLASSEX1 model with \SPSD\ba88t.cpr appear as follows:

Table 1U: Selected Quantities for Census Families by Number of children in census family

Number of children in census family	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	Family Allowance Increase (M)
0	6401.6	0.0	0.0	0.0
1	1454.2	516.5	516.5	0.0
2	1430.7	1061.7	1061.7	0.0
3	612.9	850.0	776.5	73.5
4	111.9	229.5	202.6	26.8
5	36.8	83.7	70.4	13.3
6	5.3	28.4	25.8	2.5
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
All	10053.4	2769.8	2653.6	116.2

Table 2U: Selected Quantities for Census Families by Census family type

Census family type	Family Allowance Increase (M)	Disposable Income Increase (M)
With Kids, 1 Adult	16.7	15.6
With Kids, 2+ Adult	99.4	72.3
With Elderly, 1 Adult	0.0	0.0
With Elderly, 2+ Adult	0.0	0.0
Other, 1 Adult	0.0	0.0
Other, 2+ Adult	0.0	0.0
All	116.2	87.9

The values in Tables 1U and 2U result from running the new model on the 5% subset of the SPSPD in 1988 (ba88t.cpr) and requesting the tables described above. The first table confirms that we seem to be giving the additional Family Allowances to the right kinds of census families. Increased benefits, some \$116 million of them, appear only for census families



with more than two children aged 0-17, and the gross amounts are \$120 times the number of such "excess" children in those families.

The second table offers more evidence that the new benefits are being given only to the right kind of census families and, further, that the new benefits are being partially taxed back. Moreover, the degree of tax recovery is lower for one-parent families than for two-parent families; this is to be expected since (1) those reporting Family Allowance benefits in two-parent families tend to have higher incomes and to be subject to higher marginal tax rates, and (2) the Income Tax Act requires that the higher net income spouse report the Family Allowance benefits.

We conclude from the values appearing in these tables that the changes made above have quite probably been successful in implementing our intentions.

The testing just described completes our quick start example. Because of the example's focus we have perhaps not been quite as careful and methodical as would be warranted in the case of a real application. Thus, we mention briefly here a number of things that we might have chosen to do in implementing our hypothetical change.

We might have added "revision history" comments to the files `Adrv.cpp` and `Afamod.cpp` to document the nature of the changes and our reasons for implementing them as we did. This form of documentation is an element of sound professional practice for software development and maintenance.

We might have created an intermediate (local) floating point variable in `Afamod.cpp` to store the increment in a family's Family Allowance benefit. This increment, once computed, could then have been assigned directly to the `tfa`, `ffa` and `fa` variables so that we would not have been computing the identical expression three times in parallel. Possible minor efficiency gains aside, the resulting code would probably have been slightly easier to understand.

We might have made a parameter out of the \$10 per month (\$120 per year) value, in case we wanted to repeat the analysis later for a different value of the supplementary Family Allowance benefit. Similarly, we might have made a parameter out of the number of children NOT eligible for the additional benefit; perhaps someone would want to know the impacts of restricting the extra benefits to families with four or more children, or relaxing them to admit families with only two children aged 0-17.

We might have chosen to create a new variable that would contain just the pre-tax increment for the family, making this variable part of the structure for the household so that we could more conveniently tabulate this "difference" variable in crosstabulations or export it for subsequent analysis in SAS.

We might have chosen to conduct more ambitious tests to ensure that the desired changes had been implemented. For example, we might have produced a table showing the relative sizes of the changes in federal and provincial income taxes to ensure that the new benefits were being appropriately considered at both the federal and provincial levels. We might

have tabulated the size of the change in the child tax credit to assess whether the new Family Allowance benefits were being properly taken into account in that credit's definition of income.

In general, the style of alteration and the degree of testing conducted here are appropriate for the limited goals of this introductory example. However, for a more serious glass box application the user will probably wish to be more methodical in making the necessary changes, devoting more attention to issues of documentation, labeling, validation and possibly to efficiency of computation.

## SUMMARY

This chapter has provided a first-pass description of glass box applications in the SPSM, illustrating them with a specific example. Section topics included changing the substantive calculations in a variant Family Allowance function, altering the SPSM driver function that coordinates the calculation of taxes and transfers, and using the C++ compiler to create a new version of the model. A short section on validation illustrated the generation of tables to assess the success of the change.

## SPSD/M and Glass Box Directory Structure

This chapter provides for glass box users an explanation of the hard disk directory structure within which the SPSM operates. The information it contains is relevant because it tells the user where certain items are located, which ones must be left in place untouched, which ones are designed to serve as templates for changes, which ones are to serve purely as examples for code that the user will build, etc.

Consider the following representation of user's hard disk directory structure:

```
C:  [ Root directory ]
|--- MSC    [ Microsoft C compiler, with its own subdirectories ]
|--- SPSPD  [ Data for the SPSPD/M, with no subdirectories ]
|--- SPSM   [ SPSM proper, subdirectories as shown ]
|           |--- DEFS
|           |--- EXAMPLE
|           |--- GLASS
|           |--- MODEL
|           |--- WIN32
|--- GLASSEX1 [ Glass box task subdirectory 1 ]
|--- GLASSEX2 [ Glass box task subdirectory 2 ]
etc.
```

At the top of the figure we see the user's root directory, with two first-level subdirectories MSC and SPSPD. The MSC subdirectory contains the user's compiler, absolutely necessary for the creation of glass box applications; MSC contains a number of lower level subdirectories not shown here. The SPSPD subdirectory contains all of the SPSPD/M's raw data and a number of default parameter files; it has no lower-level subdirectories.

Of more direct applicability to the glass box user is the SPSM subdirectory and its lower-level subdirectories. These were created automatically for the user during the SPSM installation; the names used here are the recommended defaults. We provide here brief descriptions of each of these directories -- their major contents and relevance to glass box

applications.

An initial, general-level comment is in order -- **THE USER SHOULD NOT CHANGE ANYTHING IN ANY OF THESE SPSM SUBDIRECTORIES.** (1) Glass box applications will always involve working with **COPIES** of some of the files in these subdirectories. (2) All of the user's glass box work will be done in one of the **SEPARATE SUBDIRECTORIES** that the user has created to contain the working files for glass box applications. It might even be useful for the user to switch on the read-only attribute for all of the files in these subdirectories.

- |         |  |
|---------|--|
| DEFS    | This subdirectory contains a number of header files that define structures and constants used throughout the SPSM. Of greatest interest to the glass box user will be the <code>vs.h</code> file that defines the hierarchical data structure that holds the SPSD/M's information about households and individuals. Recall, however, that the user will never have occasion to modify this structure. The user's addition of user-defined variables is accomplished via a COPY of the <code>vsu.h</code> file.   |
| EXAMPLE | This subdirectory contains various "INCLUDE" files that serve to specify parameters for the sample runs described in the tutorial portion of the <i>Introduction and Overview Guide</i> . Although they are potentially very useful in testing for the successful installation of the SPSM and in learning how to use models that have already been developed, these files are not directly relevant to the development of glass box models, and can be ignored for purposes of this glass box oriented discussion.  |
| GLASS   | This subdirectory contains templates that the user will use as starting points for the code that s/he writes to create variant tax/transfer systems and models. (1) It contains the source code for all of the SPSM's tax and benefit functions; the user will probably find it most efficient to create any new functions by modifying COPIES of these elements. (2) It contains functions that make the user defined parameters and variables accessible to the broader SPSM, together with associated header files that define the relevant structures to hold the user-defined variables and parameters. |
| MODEL   | This subdirectory contains examples of the definitions of model variables and parameters. The elements in the subdirectory are intended ONLY to serve as concrete examples for the user when s/he begins to define new parameters and variables for glass box applications. The user will never have occasion to modify the contents of these files, nor even to use or alter copies of the files.   |
| WIN32   | This subdirectory contains a small number of WINDOWS 32 bits "controlling object files" that govern the form of the overlay structure that the SPSM uses. At a very general level, these items are similar to those in LIB in the sense that SPSMGL.dsw needs them and knows how to use them in the compilation of a new version of the model. It also contains some executable files used in the modification of SDSD in a project.   |

At the very bottom of the representation of the user's hard disk subdirectory structure is a glass box application "task" subdirectory GLASSEX1 and two sub-subdirectory WINREL and WINDEBUG. Users may have as many such task subdirectories as are required for the glass

box applications they build. This one corresponds to the Quick Start example described in Chapter 2. It contains all of the files that the user creates in replicating that example. The specific files are as follows:

```
ADRV.CPP
AFAMOD.CPP
FAQSTST1.CPR
FAQSTST1.TBL
SPSMGL.DSP
SPSMGL.DSW
SPSMGL.NCB
SPSMGL.OPT
SPSMGL.PLG
GLASSEX1.EXE
GLASSEX1.PDB
WINREL
WINDEBUB
```

ADRV.CPP and AFAMOD.CPP are the C++ source code files copied from the GLASS subdirectory and then modified to reflect the desired new program logic; their OBJ counterparts are the object files produced as outputs when the ".CPP" files are compiled in WINDEBUB and WINREL. GLASSEX1.EXE and GLASSEX1.pdb were created by the compile command. Finally, FAQSTST1.CPR is the control parameter file for runs of the FAQSTST1 program, and FAQSTST1.TBL contains the crosstabulations that the associated run of SPSMFAQS produced.

The critical information in this chapter can then be summarized as follows:

1. **No SPSM user should change ANYTHING in the SPSM subdirectory or in any of its subdirectories created during the SPSM installation.** (Note however that certain files that may be definitely unnecessary can be deleted in their entirety.)
2. The glass box user will establish separate "task" subdirectories for glass box applications. Preferably these will not be subdirectories under SPSM.
3. **The glass box user will copy the relevant elements from the SPSM\GLASS directory, using them as templates for the changes to be made.** The changes themselves are then made to these COPIES. Subsequent sections in this *Programmer's Guide* indicate in considerable detail what the user must change and where the relevant templates are located.
4. The key subdirectories \SPSM\DEFS should all be added in **Tools: Options: Directory**, since definitions relevant to glass box applications reside there.

## The SPSD Household/Individual Structure

This chapter has three major goals, each of them developed in a separate section, but all of them relating to the general topic of the SPSD/M's data structures and their usage.

This following section provides a snapshot overview of the SPSM's framework for storing

data about the household, its families, and their component individuals. An appreciation of this structure is crucial to the glass box user as he/she seeks to refer to or alter the values of existing data variables and modeled variables, and to create such new variables as would be necessary for a customized version of the SPSM.

The second section develops the use of pointer variables as a major tool by which the user accesses individual elements of the data. It also describes the major naming conventions relevant for glass box applications. These topics are relevant both for users building their own glass box applications, and those seeking to understand the standard SPSM algorithms. The underlying "philosophy" for this development is consistent with the rest of this guide -- in many respects it is considerably more important for the glass box user to know how, mechanically, to do something in a standardized, robust fashion, than to understand all of the design-oriented reasons behind the structures and techniques. In other words, the section's focus is determinedly practical; it concentrates much more on the mechanics of "how-to" than the niceties of "why".

The third section provides a "bestiary" of code fragments for performing common glass box tasks, particularly as regards to data structures. The idea is not only that the user should be able to copy an existing wheel rather than re-inventing it, but that the copied wheel should further exist in a standardized format, and not require debugging. The section's code fragments include (a) processing relevant individuals/families via "for" statements, (b) referring to other family members, (c) accessing existing database and modeled variables, and (d) assigning new values to variables.

## **THE SPSD/M'S DATA STRUCTURE**

The SPSD is a file whose order is fixed. It cannot be sorted by the user. The sort order of the database is critical to understand when attempting to loop through households. The database is clustered into households that are randomly sorted in a stratified way. Each individual household is then sorted as follows:

Household

    Economic Families

        Census Families

            Nuclear Families

                Head of Family

                Spouse if present

                Youngest Child to Oldest Child

Within a household, individuals are grouped into economic families. Within an economic family, individuals are grouped into census families. Within the census family, individuals are grouped into nuclear families. Within the nuclear family, the head is always first followed by the spouse if present. Children then follow sorted according to their age.

An entire household is loaded into the data structure specified above. Loops may then be established to process any of the units of analysis within a household.

Detailed descriptions of the substance of individual SPSPD/M variables themselves appear in the *Variable Guide*. Much of the detail with respect to the content of the several structures can be found in `vs.h`. The key items required to define variables can be found in `spsm.h`. Some of the macros allow the user to do things symbolically to make their meanings clearer, or for consistency in numerical precision:

```
#define LOGICAL int          /* type used to store true or false values      */
#define TRUE 1              /* manifest constants to make code more readable */
#define FALSE 0
#define NUMBER float
#define ZERO (float) 0.0
#define HALF (float) 0.5
#define ONE (float) 1.0
#define THOUSAND (float) 1000.0
#define MILLION (float) 1000000.0
```

## INTRODUCTION TO POINTERS IN THE SPSPD/M

The `uv` structure is one whose contents are defined by the user, in terms of both substance and variable names. A chapter describes how the user creates new variables, e.g. defining a new tax or transfer program. The user controls the substance of "`uv`" via the `vsu.h` header file, and the `vsdu.cpp` file, but can alter the values of the defined elements themselves anywhere inside `Adrv.cpp`. These definitional and assignment capacities are the essence of glass box applications when the user needs to add new variables. Of course the user must be careful to give any new variable/tax to the right individual(s) so that roll-ups will work properly throughout the remainder of the SPSPM.

The C language makes heavy use of pointer variables, i.e. variables that point to a particular area of memory, and especially to a specific data structure. Although the portions of the SPSPM's source code dealing with tax/transfer algorithms make less use of pointers and pointer arithmetic than those portions closed to the user, the glass box user will still have to employ pointers. Even though the usage of pointers is essential, the design of the SPSPM has made it as simple as the designers could manage. A variety of macros and code fragments are provided to make the pointer usage as simple and often as mechanical as was feasible. The Bestiary section briefly shows how these pointers are applied for typical glass box tasks such as looping and referencing. Note, however, that this section is in no way intended to provide a comprehensive course in pointer usage more generally outside the SPSPM.

## THE BESTIARY

A bestiary is a "collection of descriptions of real or imaginary animals". The particular "animals" collected and described here are real. They are fragments of C-language source code likely to be useful to the glass box user as s/he reads and writes the code for tax/transfer programs. The code fragments described here are all included in the file `BESTIARY.CPP` so that the user can copy the segments without having to retype them.

The elements of the bestiary are provided in support of a philosophy emphasized throughout this guide. More precisely, users should not have to reinvent the wheel, but should be given

every assistance in taking advantage of things that already exist within the SPSM. Being able to copy existing code, perhaps modifying it in the process, provides four major advantages.

1. The existing source code is known to be correct, and thus doesn't have to be debugged.
2. There will be greater consistency between the user's code and that of the distributed SPSM.
3. Copying is much faster than re-entry.
4. The user can often get the needed job done, safely, without having to understand all of the underlying detail. The general format used is that of a heading, followed by the code itself, and, sometimes, a short comment or explanation.

### Examples of Looping:

One of the most common tasks in reading, modifying or writing code is looping through the relevant units in a household or one of its substructures. The following set of code segments probably come close to being exhaustive as regards the looping required by the user. Note that the source code segments include the relevant definitions required. E.g. in the first example below, the user must declare the pointer 'in' of type 'P\_in,' and the integer, 'ini' so that they can be used in the operation of the loop. In practice, the definitions will appear in the source code prior to the loop itself.

```
/** * PROCESS ALL INDIVIDUALS IN HOUSEHOLD hh */

register P_in in;
int ini;

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; ini++, in++) {
    DEBUG2("%s processing individual %d in household\n", ini);
    /* code here, using pointer 'in' */
}
```

In the preceding loop, and the others that follow, the C 'for' statement is used. Items before the initial semicolon initialize variables for the looping. The condition between the two semicolons specifies when the loop is to continue. The items still within the parentheses, but after the second semicolon specify the incrementing necessary for the next iteration. Also included in the code fragment is a 'code here' comment. It indicates where the SPSM's code, or the user's code, should go to act on the unit through which the loop cycles. The 'code here' comment also identifies that unit in terms of the pointer that the loop controls.

```
/** * PROCESS ALL INDIVIDUALS IN ECONOMIC FAMILY ef */

register P_in in;
int ini;
for (ini=0, in=ef->efin; ini<ef->efnpers; ini++, in++) {
    DEBUG2("%s processing individual %d in economic family\n", ini);
    /* code here, using pointer 'in' */
}

/** * PROCESS ALL INDIVIDUALS IN CENSUS FAMILY cf */

register P_in in;
int ini;
```

```

for (ini=0, in=cf->cfin; ini<cf->cfnpers; ini++, in++) {
DEBUG2("%s processing individual %d in census family\n", ini);
/* code here, using pointer 'in' */
}

/**** PROCESS ALL CHILDREN (including 18+) IN CENSUS FAMILY cf      **/

register P_in in;
int ini;
for (ini=0, in=cf->cfinch; ini<cf->cfnchild; ini++, in++) {
DEBUG2("%s processing child (including 18+) %d in census family\n", ini);
/* code here, using pointer 'in' */
}

/**** PROCESS YOUNG CHILDREN IN CENSUS FAMILY cf      **/

register P_in in;
int ini;
for (ini=0, in=cf->cfinch; ini<cf->cfnkids; ini++, in++) {
DEBUG2("%s processing child (<18) %d in census family\n", ini);
/* code here, using pointer 'in' */
}

/**** PROCESS ALL INDIVIDUALS IN NUCLEAR FAMILY nf      **/

register P_in in;
int ini;
for (ini=0, in=nf->nfin; ini<nf->nfnpers; ini++, in++) {
DEBUG2("%s processing individual %d in nuclear family\n", ini);
/* code here, using pointer 'in' */
}

/**** PROCESS CHILDREN IN NUCLEAR FAMILY nf      **/

register P_in in;
int ini;
for (ini=0, in=nf->nfinch; ini<nf->nfnkids; ini++, in++) {
DEBUG2("%s processing child %d in nuclear family\n", ini);
/* code here, using pointer 'in' */
}

/**** PROCESS ALL ECONOMIC FAMILIES IN HOUSEHOLD hh      **/

P_ef ef;
int efi;
for (efi=0, ef=&hh->ef[0]; efi<hh->hhnef; efi++, ef++) {
DEBUG2("%s processing economic family %d\n", efi);
/* code here, using pointer 'ef' */
}

/**** PROCESS ALL CENSUS FAMILIES IN HOUSEHOLD hh      **/

P_cf cf;
int cfi;
for (cfi=0, cf=&hh->cf[0]; cfi<hh->hhncf; cfi++, cf++) {
DEBUG2("%s processing census family %d\n", cfi);
/* code here, using pointer 'cf' */
}

```



```

    /*** PROCESS ALL NUCLEAR FAMILIES IN HOUSEHOLD hh      **/

    P_nf nf;
    int nfi;
    for (nfi=0, nf=&hh->nf[0]; nfi<hh->hhnnf; nfi++, nf++) {
        DEBUG2("%s processing nuclear family %d\n", nfi);
        /* code here, using pointer 'nf' */
    }
}

```

## References With Respect to an Individual:

Another common glass box task involves referring to other individuals in a structure or substructure, or to units of analysis "higher up" in the structure. It is via such references that the user can refer to characteristics such as the province of residence for an individual, the income of the spouse of the eldest member of a census family (if that spouse exists), or the age of the second oldest child living in any of the census families within a common economic family.

```

    /*** REFERENCE SPOUSE OF INDIVIDUAL in      **/

    if (in->id.idspoflg) {
        P_in inspo;
        inspo = in->id.idinspo;
        /* code here, using pointer 'inspo' */
    }

```

Notice here that there will not always exist a spouse.

```

    /*** REFERENCE HOUSEHOLD OF INDIVIDUAL in      **/

    P_hh hh;
    hh = in->id.idhh;
    /* code here, using pointer 'hh' */

```

With the pointer to the household retrieved, the user then has access to household characteristics such as province of residence. In contrast to the situation with the spouse of an individual, the household will always exist.

```

    /*** REFERENCE ECONOMIC FAMILY OF INDIVIDUAL in      **/
    P_ef ef;
    ef = in->id.idef;
    /* code here, using pointer 'ef' */

```

Similarly, the individual's economic family will always exist, and will be relevant for ascertaining whether the individual lives in a below - LICO unit.

```

    /*** REFERENCE CENSUS FAMILY OF INDIVIDUAL in      **/
    P_cf cf;
    cf = in->id.idcf;
    /* code here, using pointer 'cf' */
    /*** REFERENCE NUCLEAR FAMILY OF INDIVIDUAL in      **/
    P_nf nf;
    nf = in->id.idnf;
    /* code here, using pointer 'nf' */

```

These key references, coupled with the looping fragments of the previous section, permit the user to do, relatively conveniently, almost anything likely to be needed for tax/transfer simulation.

## SUMMARY

The first part of this chapter described the data structure used for SPSD/M. That part also identified the most important manifest constants and function macros the user will encounter in the SPSM's source code. The later portions described the role of pointer variables in the SPSM and characterized the major pointer types used. They concluded with a bestiary of code fragments for common glass box tasks, looping through individuals and family units, and referring to an individual's spouse or to the units of analysis that contain him/her.

The next chapter builds on this foundation by describing how the SPSM processes households in terms of calculating taxes and transfers. That description is in turn a foundation for the later chapters that indicate how to add user-defined parameters and variables in the course of modifying the logic of the tax/transfer system.

## SPSM Function Calling Structure

The calculation of taxes and cash transfers for a household is controlled by a function whose only task is to call all other individual tax/transfer algorithm functions. The sequence of calls is critical to the simulation due to the informational requirements of the tax/transfer functions. For example, net income must be known before GIS can be calculated. The following list gives the functions called by `drv` and `adrv` in the order in which they are called.

Function	Description
<code>ui(hh)</code>	Compute Unemployment Insurance benefit
<code>famod(hh)</code>	Compute family allowances
<code>oas(hh)</code>	Compute old age security
<code>dem(hh)</code>	Compute new demogrants
<code>txinet(hh)</code>	Compute net income
<code>gis(hh)</code>	Compute guaranteed income supplement for elderly
<code>gist(hh)</code>	Compute provincial elderly top-ups
<code>samod(hh)</code>	Compute social assistance
<code>txitax(hh)</code>	Compute taxable income
<code>txhstr(hh)</code>	Compute child & spouse deductions
<code>txcalc(hh)</code>	Compute federal tax
<code>txctc(hh)</code>	Compute child tax credit
<code>txfstc(hh)</code>	Compute federal sales tax credit
<code>txprov(hh)</code>	Compute provincial taxes and credits
<code>gai(hh)</code>	Compute new guarantees, refundable credits
<code>memo1(hh)</code>	Compute disposable income, etc.
<code>ctmod(hh)</code>	Compute commodity taxes and allocate to persons
<code>memo2(hh)</code>	Compute consumable income, etc.
<code>cceopt(hh, drv)</code>	Zero CCE for young kids if optimal
<code>classu(hh)</code>	Compute user-defined reporting variables (in \glassbox)

The calling order of the component functions of `drv` reflects the logical precedence

between them.

- The first functions, `ui`, `famod` and `oas`, simulate programs whose benefits are determined by factors other than income and as such are called first.
- `dem` is a stub routine for glass box applications that require calculations to occur before entering the tax system routines.
- `txinet` calculates net income prior to certain transfers.
- `gis` calculates transfers to the elderly.
- `samod` calculates social assistance or guaranteed income transfers.
- Federal and provincial taxes are calculated next in the next four functions with the `tx` prefix (`txitax`, `txhstr`, `txcalc`, and `txprov`).
- `gist`, `txctc`, and `txfstc` calculate income tested transfer programs.
- `gai` is another stub routine that is intended for use by glass box users who wish to simulate options requiring information on all personal income taxes and cash transfers. For example, users may use this function to simulate an income supplementation program.
- The `memo1` and `memo2` functions create aggregate variables for reporting.
- In the `ctmod` function, sales and excise taxes are calculated by applying Input/Output based effective sales tax rates to observed family expenditures.
- `cceopt` optimizes income by maximizing the childcare expense credit and the child tax credit.
- `classu` is a stub routine that allows the glass box user to compute and assign values to new or re-defined variables.

The functions called by `drv` call other functions and sub-functions in order to complete their calculations. The following page contains a complete list of the names of functions and sub-functions along with a short description in the order in which they are called by `drv`. Please refer to the specific function in the *Algorithm Guide* for a more detailed description. Sub-functions can be found listed under the function that calls them. Thus for a complete understanding of the calculation of net income one would have to consult both the `txinet` and `txcea` functions.

Function names are printed in lower case, bold, courier font (e.g. `txinet`, `txcalc`) and correspond to a single C language source code file (e.g. `tixnet.cpp`, `txcalc.cpp`). Sub-functions are defined within the function (file) that calls them and are shown in lower case, courier font (e.g. `uisqz`, `gissub`). The following example is a call of a sub-function `uiclm()` in `ui.cpp` where `uiclm` is defined in a section of `ui.cpp`.

```
valid_claim = uiclm(in, &in->id.uc1, in->id.uc1.ucy1, &in->im.ub1,  
                  hh->hd.hdprov, hh->hd.hdurb, wctb);
```

## Glass Box Development: Adding Typical Scalar Parameters

As its title suggests, this chapter explains to the glass box user the mechanics of the programming tasks associated with adding typical scalar parameters during the development of glass box applications. Structurally, the chapter communicates this information via a

detailed worked example. The first section reviews the general procedure for developing glass box applications, describing the steps that are fundamental to any model alteration, be it changing code, adding parameters or adding variables. The second section takes up several preliminaries to parameter addition. It also describes the nature of the example to be used, an extension of the Family Allowance supplement example used in this Guide's Quick Start Chapter. The remaining sections then use the example to explain in detail the steps involved in adding the most common kinds of scalar parameters to a model. Finally, the last section summarizes the key points regarding the addition to a model of these common forms of parameters.

## **GENERAL PROCEDURE FOR MAKING GLASS BOX CHANGES: A RECAPITULATION**

The previous section has already described the general procedure for developing glass box applications, including the reasoning behind the steps. We summarize the key points here in capsule form.

- Create Task Sub-directory
- Identify Files to be Changed
- Copy relevant Files to Task Sub-directory
- Edit Relevant Files
- Compile the new version
- Test the New Version of the Model
- Carry Out the Intended Analysis

### **Create Task Sub-directory**

The user creates a new "task subdirectory" to hold the files relevant for the new glass box application. She/he will edit files in the task subdirectory, leaving all of the other SPSPD/M files alone.

### **Identify Files to be Changed**

The user identifies those files in `c:\spsm\glass` for which variants will have to be created. For example, in the Quick Start example, we identified `Afamod.cpp`, `Adrv.cpp` and `SPSMGL.dsw`. The example appearing in this chapter indicates how other files, e.g. `Mpu.h` and `Ampd.cpp`, are relevant to adding new parameters to a glass box application. A section will explain how still other files, `Vsu.h` and `Vsdu.cpp`, are relevant when the user wishes to add new variables to a model. Clearly, the tax/transfer function files that use the new parameters must also be changed. At times, the user may find it more efficient to use files already developed in a previous application as templates, rather than going all the way back to the glass subdirectory's template files.

### **Copy Relevant Files to Task Sub-directory**

The user copies all of the identified-relevant files across to the task subdirectory. The user will work only with these copies, leaving the originals unchanged.

### **Edit Those Relevant Files**

The user makes appropriate changes in each of the files identified as relevant. We recommend that the changes be made in the following order:

1. Include all relevant files into the project and change the output file name in Project: Setting: Link.
2. Edit the `Adrv.cpp` file, as necessary.
3. Edit the `Mpu.h` and `Ampd.cpp` files, when appropriate, to add any new parameters to the model.
4. Edit the `Vsu.h` and `Vsdu.cpp` files, as appropriate, to add any new output variables to the model.
5. Edit the source code files to add the desired new substantive logic to the tax/transfer system.

We shall follow this prescribed order in the examples we present in this and subsequent sections.

### **Compile the new version**

The user should activate the Debugging setting in Build: Set Active Configuration and then run a debug execution of the project. When the program changes are properly implemented then the new model should be compiled.

### **Test the New Version of the Model**

The user tests the new version via a set of validation analyses designed to reveal any problems with the logic that has been added or modified. This step may require going back to some of the earlier ones to remedy any deficiencies that are discovered.

### **Carry Out the Intended Analysis**

Finally, once the validation is complete, the user can proceed with "production runs" of the new executable code to simulate the consequences of the change that was modeled.

## **INTRODUCTION TO PARAMETER ADDITION**

This section takes up a few critical preliminaries to the procedure for adding typical scalar parameters. First, it illustrates why a user might wish to add one or more parameters to a model. In addition, it describes the substance of the new parameters we use to illustrate the addition of typical parameters.

As noted at the end of the Quick Start example, our hypothetical analyst there took a few shortcuts that might be done differently in a real-world policy development exercise,

especially if the new model was intended to be used repeatedly or by multiple analysts. One of these shortcuts was to "hardwire" the \$120 per year Family Allowance increment right into the `Afamod.cpp` function. Although this might be acceptable if the user would never want to try another value for the increment, it is not particularly efficient should there be any interest in examining the impacts of other values. The user would need to re-edit the code and then to recompile the model for each separate value to be examined; the user might, for example, seek to confirm a belief that the impacts are generally proportional to the amount of the increment, and wish to try multiple values by way of investigation. With appropriate parameters added to the model, no additional editing is required, and the user can investigate multiple values without re-compilation by simply supplying new parameter values to the modified model.

Consequently, several sections in this chapter describe the steps necessary to add new parameters to the model, cleaning up the Quick Start example by way of a specific illustration. This chapter restricts itself to the most commonly used forms of scalar parameters. We believe that the kinds of additions described here will meet perhaps 80% of the parameter addition needs of glass box users. We leave the definition of more esoteric scalar parameters, and of vectors and matrices of parameters, to the last sections. Whatever the type of new parameters, once added to a model, they are available to all functions called by `Adrv.c`; they are not restricted to the function for any single transfer program.

Substantively, we shall add three parameters to a variant of the Quick Start model. The three additions correspond to the three most common forms of parameters that glass box users will have occasion to use.

1. The first parameter, a scalar "float" or "real" value, will provide the value of the Family Allowance increment given in respect of certain children; it will eliminate the hardwired \$120.00 value. We'll call this parameter `fasuppc` (Family Allowance Supplement Per Child).
2. The second parameter, a scalar integer value, will indicate the number of children at which the supplement begins to be payable; it will eliminate the hardwired value of "3" used in the Quick Start example. We'll call this parameter `fasupfec` (Family Allowance Supplement's First Eligible Child).
3. The third parameter, a "flag" variable that is effectively a boolean switch, will indicate whether any attention is to be paid to the first two parameters. In this, its function parallels that of the many "flag" variables used throughout the SPSM. When turned "on" it will enable the computation of the supplement; when turned "off" the model will calculate Family Allowances with no provision for the supplement. We'll call this parameter `fasupflg` (Family Allowance Supplement Flag).

Our description assumes that the user has chosen to use `\glassex2` as the task directory, creating it if necessary.

**COPY FILES `ADRV.CPP`, `MPU.H`, `AMPD.CPP`, `AFAMOD.CPP`, `SPSMGL.DSW`**

The user copies to the new task subdirectory all of the files for which changes are required..

Similarly, the user will wish to modify `Adrv.cpp` to update the description used for the substantive files (here only `Afamod.cpp`) being changed. Thus, `Adrv.cpp` needs to be copied.

Two other files, `Mpu.h` and `Ampd.cpp`, are always relevant when the user wishes to add a new model parameter. `Mpu.h` (Model Parameters, User) is a C language header file that defines the nature of the new parameter. `Ampd.cpp` (Alternate Model Parameter Definitions) contains the function invocations that make the user's parameters known throughout the rest of the SPSM, e.g. so that they can be referenced by name for purposes of changing values "on the fly" when the user executes an SPSM executable file.

The user must copy these `Mpu.h` and `Ampd.cpp` files across from the glass subdirectory or some equivalent source. If, for example, the user has already, elsewhere, modified these files to define other parameters, and wishes to retain those previous modifications, s/he can copy templates for `Mpu.h` and `Ampd.cpp` from the subdirectory in which they exist. By the term "templates" we refer to existing files, or pieces of text or code, that serve as a convenient starting point for making any desired modifications. For example, it would make no sense at all for the user to enter, from scratch, completely new versions of the relevant files. In this example, we'll assume that these are the first parameters being added, and will copy the templates from glass.

Finally of course, the user must copy the substantive tax/transfer function or functions that will use the new parameter. For our purposes the only relevant substantive function is the `Afamod.cpp` function. Rather than copying it from glass and then having to start from scratch, we'll copy it from `glassex1` so that some of our work is already done, e.g. locating where the assignment of the increment should be made.

**The user will have to copy SPSMGL.dsw that describes the project environment.**

## **UPDATE THE PROJECT**

All the required files should be included in the project and the name of the output executable changed in Project: Setting: Link to `glassex2.exe`.

## **UPDATE THE ALGORITHM DESCRIPTION IN ADRV.C**

Recall from the Quick Start example that the `altname[]` and `Tdrv[]` global variables received new values to reflect and document the nature of the changes to be made. Here, with a new version of the model being created, a corresponding substitution is in order. The two substitutions, consisting exclusively of the contents of the two strings, result in the following code:

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "Parameterized FA Supplement";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "Parameterized FA Supplement"
```

At this point we can carry out a debugging compilation to check our modification. Such a check helps a user to identify syntax errors while the nature of the modification is still fresh in the memory. To do so, select Win32Debug project in Project:Set Active Project and then

do Build:Start Debug. If compilation and links are required, C++ will let you know.

## **MODIFY MPU.H TO DEFINE THE NEW PARAMETERS**

The user next needs to change the file `Mpu.h` to define the type of the new parameters. When the change is made in the glass version of `Mpu.h`, the line containing the string "UMDUMMY" is replaced with definitions of the new parameter(s). The name "UMDUMMY" refers to "User Model Dummy parameter." We're calling the first new parameter FASUPPC to indicate that it is the amount of the FA supplement per relevant child. Before the change the indicated line (about line 62) reads:

```
int UMDUMMY;          /* dummy entry                                */
```

Because, as the label indicates, this entry is only a placeholder, dummy, entry so that the SPSM will have something to work with if the user has not yet defined any user parameters, we delete this line completely. We replace it with the lines:

```
NUMBER FASUPPC;      /* Family Allowance Supplement per Child */
int     FASUPFEC;     /* FA Supplement, First Eligible Child */
int     FASUPFLAG;    /* FA Supplement, Activation Flag      */
```

In the first line, "NUMBER" is a macro used by the SPSM to ensure portability across machines; it corresponds to the type "float". FASUPPC is the name of the new parameter. The SPSM convention is such that parameter names are capitalized. The other two parameters are naturally integers. For readability, we have also added comments on the right to indicate the nature of the parameter values.

These simple additions complete our changes to `Mpu.h`. Typically, if we were adding new parameters to a non-empty set of user parameters already in place, we would simply add the new definitions to the bottom of the existing list in `Mpu.h`, just as the FASUPFEC and FASUPFLAG parameters here follow the FASUPPC parameter.

The SPSM allocates space for up to 500 such new parameters, easily enough for typical glass box user applications. Even more parameter additions are possible when some of them are of the smaller "int" type. **Any attempt to exceed this limit will result in a compile-time error message that will make the problem apparent.**

## **MODIFY AMPD.CPP TO MAKE THE PARAMETERS AVAILABLE TO THE SPSM**

The user also needs to change the `Ampd.cpp` file to make the new parameter "visible" throughout the portions of the SPSM that may need to reference it. The SPSM provides a function "pmaddent" (Parameter Module, Add Entry) to carry out this task. The user calls the function once for each new parameter, just before the "DEBUG\_OFF (Ampd)" statement near the end of `Ampd.cpp`, at about line 138.

If the user is working on a copy of `Ampd.cpp` that already contains invocations of `pmaddent` for other parameters, those other calls can be used as templates. In our example though, since there are, as yet, no other parameters added, we copy a `pmaddent` template from the file `C:\SPSM\MODEL\Mpd1.cpp` (Model Parameter Definition File 1). For our first parameter, FASUPPC, we recognize that this NUMBER type parameter should be very similar to the STDFA parameter appearing at about line 252. We simply copy that `pmaddent`



invocation and make appropriate substitutions. This, practice, copying something generally similar that already exists and works, and then modifying it, is standard practice in glass box development. The invocation, as copied, looks like:

```
pmaddent(pcp, "STDFA", (char *)&MP.STDFA, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

We modify it for our purposes by changing the two references to STDFA to correspond to our new parameter. Replacing "STDFA" by "FASUPPC" and "(char \*)&MP.STDFA" by "(char \*)&MP.UM.FASUPPC", because the new parameter is an element of the substructure UM (User Model) that lies within the MP (Model Parameters) structure, we obtain the result:

```
pmaddent(pcp, "FASUPPC", (char *)&MP.UM.FASUPPC, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

For the moment we simply retain all of the other arguments to the function without having to worry about what they represent. As long as we have chosen an appropriate template to steal from, there is no problem. Later, we'll look at the meaning of each of the arguments to pmaddent so as to facilitate more informed judgements about appropriate sources for pmaddent templates, and more effective recovery from any incorrect choices.

We choose UIWAITWKS (the integer number of weeks in the Unemployment Insurance waiting period) as our template for our integer parameter specifying the "position" of the first child in the family to be granted the supplement. Similarly, we choose an existing flag parameter to serve as the template for our new FA supplement flag; FAFLAG, which controls whether Family Allowances are computed at all, seems a good choice. Before our modifications, these two invocations appear as follows:

```
pmaddent(pcp, "UIWAITWKS", (char *)&MP.UIWAITWKS, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
pmaddent(pcp, "FAFLAG", (char *)&MP.FAFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

As with the FASUPPC parameter above, we modify each of these templates in two places, substituting the name of the parameter and its relation to the MP structure. The modified pmaddent invocations appear as follows:

```
pmaddent(pcp, "FASUPFEC", (char *)&MP.UM.FASUPFEC, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
pmaddent(pcp, "FASUPFLAG", (char *)&MP.UM.FASUPFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

These simple additions complete the modification of Ampd.cpp as regards making the VALUES of the new parameters available throughout the SPSM, once we have somehow assigned those values. Later in this section we address some of the mechanisms by which the user can make the assignments. However, we still need to provide clear labels for the parameters so that the SPSM can use them to give meaningful documentation of the model parameters as appropriate.

Once again, the design of the SPSM renders our job easy. There is a ready-made function, stradd, to implement the labeling. Just after the pmaddent statements we insert three lines to invoke this function, stradd --

```
stradd("FASUPPC", "Family Allowance Supplement per Child");
stradd("FASUPFEC", "FA Supplement, First Child Payable");
stradd("FASUPFLAG", "FA Supplement, Activation Flag");
```

The stradd (String Add) function, when executed, "attaches" the descriptor string to the

parameter so that the descriptor will automatically appear in all relevant SPSM documentation and labeling. With the (**stradd**) function's arguments this simple, i.e. one string identifying the name of a new parameter, with a second string providing the associated description, we do not even need to resort to a template.

The final item within this step, partial compilation of the `Ampd.cpp` function, is optional, but we recommend it as conducive to the orderly development of glass box applications. This type of partial compilation enables the user to have the compiler check for syntax errors while the nature of the modifications is still fresh in one's mind. It does not ensure that the modified source code meshes with the rest of the SPSM. Note that one has to have modified any relevant header files, here the `Mpu.h` header file, first in order for the Debug compilation to work.

### **MODIFY THE FUNCTIONS THAT USE THE NEW PARAMETER(S)**

To complete the programming changes involved in adding the parameter, we next need to alter the `Afamod.cpp` function so that it makes use of the new symbolic parameters rather than the "hardwired" values that appeared in the Quick Start example. We begin by adjusting the label defined for the function; more specifically we modify the code defining the label so that it reads --

```
/*global*/ char FAR Tfa[] = "Afamod.cpp Parameterized"
```

With this label supplied, the SPSM can use it whenever it has occasion to use the function's description in its documentation.

The substantive changes to the `Afamod.cpp` function are simple to implement.

Where the Quick Start example used "120.0", we substitute the symbolic representation "MP.UM.FASUPPC". This naming convention, exactly identical to the one used in the "pmaddent" function invocation in the `Ampd.cpp` change above, reflects FASUPPC's location within the UM (User Model) substructure of the MP (Model Parameter) structure that the SPSM uses to store all of the model parameters.

Where the Quick Start example used 3 to represent the number of children required in the family for the supplement to be paid, we substitute MP.UM.FASUPFEC. All relevant formulae are adjusted accordingly.

We make the calculation of the supplementation, and its addition to the `fa`, `tfa`, and `ffa` variables conditional on the value of the new flag variable, `fasupflag`.

Thus, the key Quick Start example source code that appeared as:

```
/* $120/yr bonus for 3rd and subsequent children <18 */
if (nch >= 3) {
    tfa += (nch-2) * 120.0;
    ffa += (nch-2) * 120.0;
}
```

becomes, in its `glassex2` incarnation:

```
/* Conditionally add a Family Allowance bonus for the
```

```
"FASUPFECth" and subsequent children <18 in the unit */
if ((MP.UM.FASUPFLAG == 1) & (nch >= MP.UM.FASUPFEC)) {
tfa += (nch-MP.UM.FASUPFEC+1) * MP.UM.FASUPPC;
ffa += (nch-MP.UM.FASUPFEC+1) * MP.UM.FASUPPC;
}
```

The underlying logic remains unchanged, but now it is specified parametrically. In addition we have modified the comment to reflect the generalization to symbolic parameters. In writing the source code in this fashion, we have trusted that users of the model will supply only reasonable values of the parameters. For example, we trust here that no user will inadvertently supply a value of zero (0) for MP.UM.FASUPFEC and unintentionally create a Family Allowance supplement for those families with zero children aged 0 through 17. Later, we'll show how the user can use the SPSM's edit-check facilities to guarantee that the parameters values are reasonable.

Once again we perform a Debug compilation to catch any syntactic errors before compiling the new model.

## **VALIDATE AND MAKE BLACK-BOX PRODUCTION RUNS**

As with the Quick Start example, we still need to test the new variant of the model to ensure that it gives reasonable results. With SPSM runs being essentially free, and not terribly time consuming, two particular validation runs immediately suggest themselves.

1. The first is a run with the FASUPPC parameter set to zero, using the same tables generated in the Quick Start example. For this run we set the FASUPFEC parameter to 3, and the FASUPFLAG parameter to 1. We expect that there will turn out to be no differences between the base and variant systems because the zero value for the parameter renders the change nil.
2. We modify the first test to supply a value of 120.0 for the FASUPPC parameter, leaving the FASUPFEC and FASUPFLAG parameters at 3 and 1. Again we request the Quick Start tables as output, expecting to observe the same results we obtained from the original Quick Start example with its hardwired 120.0 value.
3. We modify the FASUPFEC to take on a value of 2, expecting that this will considerably increase the cost of the hypothetical option, since there are relatively many two-child families. The specific tables allow us to ascertain easily, at least for the gross amount of the supplement, whether the right amounts of supplement have been calculated for each of the family types by number of children.
4. Finally, we add a fourth test to turn the supplement off via the FASUPFLAG parameter. In making this validation test, we leave the FASUPPC and FASUPFEC parameters at 120.0 and 2 so that we can be sure that any effect is caused by resetting the flag parameter to zero. As with the first validation run described above, we expect that there will be no differences between the base and option Family Allowances, the computation of the supplement having been suppressed.

For carrying out the validation tests, it remains only to assign the desired values to the new

parameters. The design of the SPSM makes this easy. If we simply run the new model without having bothered to specify a needed parameter value, the SPSM notes the omission, allowing us to provide the value via the "on-the-fly" parameter editing facility. Or, to be functionally equivalent, we could have placed an appropriate entry in the MPR (Model Parameter) file, since such files hold model parameters generally, whether the parameters are defined by the user or are built into the SPSM as distributed. Similarly, the new parameter file could have been specified in an MPI (Model Parameter Include) file. Authoritative descriptions of these latter two methods may be found in *User's Guide*.

Upon making the tests described above, we are encouraged that our change, the addition of the three new parameters, has been properly implemented because all of the sets of outputs appear as anticipated. The results of the third test, where we shift the FASUPFEC (first eligible child) parameter, are especially important. There we can check to see if appropriate amounts of supplement benefits are added to families classed by number of children aged 0 to 17. Now, with the model changes validated, we are ready to make the relevant set of production runs. For example, a client might ask us to use a FASUPPC parameter value of 60.0 to confirm our his expectation that the same number of families would be affected as with a value of 120.0, and that the costs, in aggregate and as an average per affected family, would be only half as great as for that 120.0 value. Similarly, we might substitute a much larger value, say 5000.0, to confirm our expectation that, with such a large transfer, the proportion of the supplement recovered through the tax system would rise somewhat as some families move into higher tax brackets.

## **SUMMARY/CONCLUSION**

It is useful to conclude by highlighting, but without any redevelopment, the key points relevant for adding typical scalar parameters to a model. In noting these points, it is taken as given that the analyst is working with COPIES of the relevant files, and is performing all of the modifications in a task subdirectory dedicated to the analysis at hand. We also assume that the user has updated the project to include all of the relevant source code files. In terms of technique, we assume that the user will most often be grabbing a chunk of similar existing code as a template, and then modifying it as required.

1. Modify the `Mpu.h` header file, adding one statement for each new parameter. The statement indicates the name of the parameter and its type, with NUMBER used for float values.
2. Modify `Ampd.cpp` source code file, adding two statements for each new parameter.
  - Add one "pmaddent" invocation for each parameter so that the SPSM can make its value available to all functions called by `Adrv.cpp`. Normal practice is to copy the invocation from an existing invocation and then modify it in two places -- the name of the parameter and its address.
  - Add one stradd invocation for each parameter so that the SPSM attaches the parameter's label to that new parameter.

3. Modify the relevant substantive function(s) to make use of the new parameter(s), changing the labeling as well as the internal logic of the function.
4. Debug and Compile the new model . Make the necessary "production runs" of the model and then interpret the results.

## Glass Box Development: Adding Less Typical Parameters

This chapter describes in greater detail the arguments for the `pmaddent` function and that function's use when the user adds scalar, vector and matrix parameters to glass box applications. To do this, it builds on the foundation established in the previous section(Adding typical scalar parameters), developing the new considerations for less typical scalar parameters, for vectors and lookup schedules, and for matrices. Finally, the last section summarizes the key points for regarding the addition, to a model, of these less common forms of parameters.

The first section of this chapter presents the set of arguments for the key `pmaddent` function, describing the key features of each of them. The following section then presents a list of the types of scalar parameters the user might wish to add. For each type, it indicates briefly the purpose of that specific type, describes the key `pmaddent` arguments for the type, and identifies an appropriate `pmaddent` template to use when creating a parameter of that type. Also included are sections that takes up the special considerations involved in adding vectors of parameters, following with schedule "lookup" parameters, and matrices of parameters.

### **PMADDENT: THE FUNCTION AND ITS ARGUMENTS**

Recall from section on description of adding typical parameters that the most complicated aspect of making a new parameter available to a model lies with the changes to `Ampd.cpp`, the changes to `Mpu.h` being very straightforward definitions of the parameters' types. Within the `Ampd.cpp` changes, the only significant challenge, and not by any means a particularly onerous one, comes from the invocation of the `pmaddent` function. We noted that the glass box user can usually sidestep the complexities of that function simply by choosing an "appropriate" template invocation, one copied from an "appropriately similar" parameter already defined. In this section we explain more fully the sense of the various `pmaddent` arguments, so that the glass box user will be able to use the `pmaddent` function confidently, even when there is no obvious template to be copied and modified.

Our starting point for the description of the `pmaddent` arguments is the explanatory comment that appears in `Ampd.cpp` itself (at about line 150 of the GLASS version). We'll take up each of the ten arguments in sequence. We emphasize, however, that the user should have relatively little occasion to require this information. Most of the time, the parameter to be added will be well understood, and an appropriately similar template parameter readily identifiable. In all those cases the user should simply modify the relevant templates and get on with the modeling, leaving the intricacies of `pmaddent` to those doing non-standard tasks.

`Ampd.c`'s summary of the `pmaddent` arguments is as follows:

```

/**
 * pmaddent(
 *   pcp,                <= parameter chain being extended (leave as is)
 *   "XXXXX",           <= name by which the parameter will be known
 *   (char *)&MP.UM.XXXXX, <= address of the parameter
 *   Format,            <= printing information for the parameter
 *   Agg_Type,          <= Aggregate type (scalar, vector, etc.)
 *   C_Type,            <= C-type (integer, number, string)
 *   Edit,              <= Edits to be performed
 *   Row_max,           <= Maximum number of rows, or option edit limit.
 *   Rows_addr,         <= Address of int holding current number of rows
 *   Limit,             <= Number of columns);
 */

```

The first argument (pcp) is particularly straightforward; the user ALWAYS enters the variable pcp. The argument identifies the specific parameter chain that the user is extending. Although the SPSM employs other parameter chains in its operations, the user may add parameters ONLY to the pcp chain.

The second argument, characterized by the "XXXXX" placeholder in the comment, is the user's name for the parameter. The name here will be the same one that the user employed in the Mpu.h definition. Users should be careful to choose reasonable mnemonics for these names, e.g. the FASUPFLAG name we used previously. The SPSM convention is that these names should start with an upper-case letter and should contain only upper-case letters and digits.

The third argument, characterized by the (char \*)&MP.UM.XXXXX placeholder, is the address for the parameter. The initial (C language "cast") portion of the argument, '(char \*)' is invariant. Similarly, the 'MP.UM' portion is invariant because the user's parameters are always added to the "Model Parameter, User Model" structure. The 'XXXXX' portion represents the name of the user's parameter; it is set to the string used as the second argument, but without the delimiting quotes. Finally, reflecting C's treatment of variable's addresses, the ampersand (&) is present if the parameter is a scalar, and typically absent if it is not (i.e. absent if the parameter is a vector, lookup parameter or a matrix). The common C-language device of specifically referring to the first element of an array is taken up later as a special topic. For the special case of a 'DUMMY' parameter, described below, this third argument takes on the value of 'NULL'.

The fourth argument, characterized in the description above as 'Format', is a string. It contains information about how the SPSM should display the value of the parameter when documenting it. Typically, the user will use the predefined format 'NULL', indicating that the SPSM is to print the parameter as it sees fit. Another predefined format, "F\_FRACT", contains the string "8.5" and is particularly suited for printing out the value of a fraction. The user can also enter an explicit string for the argument; e.g. using "8.0" specifies that the value should occupy 8 characters, and that it should not include a fractional part. An argument of "7.2" would specify a string occupying 7 characters, with two digits beyond the decimal point. When appropriate, e.g. for the lookup style parameters, the argument can include multiple format indicators, e.g. "8.0 8.2 8.2". The predefined format F\_LKTUR, used for P\_LKPXY type parameters provides a concrete example of this usage.

The fifth argument, characterized in the description above by 'Agg\_Type', indicates the type

of the parameter. This argument reflects a forced choice among the six integer values 0 through 5. Each of the six values has a mnemonic counterpart that the user can employ, for clarity, in place of the numeric value itself. The six values, their mnemonic counterparts, and their interpretations are as follows:

The value 0, represented mnemonically by P\_SCL, is the most common value. It is used for a parameter that is a scalar value (integer, float, fraction, etc.).

The value 1, represented mnemonically by P\_VCT, is used when the parameter is a vector. Other key information about the vector, e.g. the number of elements it contains, is given by other pmaddent arguments.

The values 2 and 3, represented by the mnemonics P\_LKPHY and P\_LKPSL, are used within the SPSM for two special kinds of schedules in which lookups are performed, one with an X-Y format and the other with a range-slope format. In the event that the user wishes to create parameters of these types, the GISST and FTX parameters provide operational examples. These two parameter types define schedules that correspond to functions LKUP1 and LKUP2 respectively; the LKUP1 and LKUP2 functions themselves are documented in the *Algorithm Guide*. The use of schedules in the SPSM is documented more fully in this chapter. The value 4, represented by the mnemonic P\_TBL, is used when the parameter is a two dimensional matrix (table). Other key information about the matrix, e.g. the numbers of rows and columns, is given by other pmaddent arguments. The commodity tax matrix CTTXRM provides a good example.

The value 5, represented by the mnemonic P\_DUMMY, will not generally be used by glass box users. This parameter type corresponds to a dummy entry used to hold the name of a header string for documentation purposes.

The sixth argument, characterized in the description above by 'C\_Type', indicates the type of the parameter. There are three possible entries for this argument. The value C\_INT is appropriate when the parameter value is inherently an integer, i.e. consists of a number with no fractional part, and has a value within the C language's bounds for integer values. The user will employ a value of C\_INT for this argument when the Mpu.h entry for the parameter used an 'int' declaration. Parameters that are "flags" or "options" will naturally be integers.

The value C\_NUM is appropriate when the parameter value may have a fractional part, or when it is too large to be stored as an integer. The user will employ a value of C\_NUM for this argument when the Mpu.h entry for the parameter used a 'NUMBER' declaration.

The value C\_STR is used when the parameter value is a dummy entry used for a header string. Glass box users will not generally have occasion to use C\_STR.

The seventh argument, characterized in the description above by 'Edit', indicates the edit checks to be imposed on the value of the parameter. The activation of these edit checks will force the value of the parameter to obey various constraints that may be appropriate. In addition, they may constrain a user's ability to modify the parameters' values at execution

time via the SPSM's parameter editing facilities. The `pmaddent` argument governing such edit checks is an integer value. Typically, the user will choose a value by entering an element from a set of predefined mnemonic values (described below).

The codes and their interpretations are as follows:

`E_NONE` (value 0) indicates that no edit checks are to be performed on this parameter.

`E_FIXL` (value 1) applies only when the parameter is a vector, lookup table or array (and thus has a known maximum number of rows). This edit code prevents the user from attempting to change the actual number of rows from the maximum value. The mnemonic here indicates that the row limit is regarded as fixed.

`E_FLAG` (value 2) indicates that the parameter is a flag. Under SPSM conventions, this means that the parameter is treated as a binary variable (defined as an integer) that must take on either the value 0 (zero) or the value 1 (one).

`E_FRCT` (value 4) indicates that the parameter is a fractional value that must fall in the domain 0.0 and 1.0, inclusive.

`E_NOCH` (value 8) indicates that the user is not allowed to make any changes to the value of the parameter via the SPSM's built-in parameter editor. This edit check can apply to any of the types of parameters, `C_INT`, `C_NUM` or `C_STR`.

`E_OPT` (value 16) indicates that the parameter is of a special "option" type, corresponding to a forced (integer) choice of values from 1 to the maximum option number permitted. The maximum number itself is provided, for option parameters, by the eighth `pmaddent` argument.

Should multiple codes be relevant, the user can simply add the relevant component values together. E.g. a value of 12 indicates a parameter that must be a fraction, and that the user is not permitted to edit dynamically at run time.

The eighth argument, characterized in the description above by '`Row_max`', indicates the maximum number of rows for certain types of parameters (`P_VEC`, `P_LKPXY`, `P_LKPSL`, or `P_TBL`). (Note however, the SPSM's flexibility, in that the actual number of rows used in a specific application may be less than this maximum.) For the other parameter types (`P_SCL` and `P_DUMMY`) this argument should take on a value of 0 (zero), except for `OPTION` parameters, where it indicates the number of legitimate option values. (A value of `N` for an `OPTION` parameter indicates that the legitimate values range from 1 to `N` inclusive.) Since scalar parameters (`P_SCL`) are the norm, this argument will most often take on the value 0.

The ninth argument, characterized in the description above by '`Rows_addr`', contains the address of the integer variable corresponding to the current (actual) number of rows for certain kinds of parameters, `P_VEC`, `P_LKPXY`, `P_LKPSL`, and `P_TBL`. When the number of rows is irrelevant, e.g. for a scalar or `DUMMY` parameter, the user enters a value of '`NULL`' for this argument; thus, this argument will typically take on the '`NULL`' value.



The tenth and last `pmaddent` argument, characterized in the description above by 'Limit', indicates, for parameters of type `P_TBL`, the number of columns in the table. In contrast to the flexibility provided for rows, where the actual number of rows may be smaller than the maximum number, the SPSM requires that the actual number of columns be fixed beforehand. For all other parameter types, this argument takes on the value of 0 (zero).

## CHARACTERIZING SCALAR PARAMETERS

With the description of `pmaddent`'s arguments complete, we turn first to the kinds of scalar parameters that the user may wish to add. The discussion here treats them in roughly descending order as regards expected frequency of use. For each of the types the description indicates (1) the general nature of the parameter, (2) the key `pmaddent` arguments, and (3) an appropriate `pmaddent` template. Even though this chapter deals primarily with more specialized types of parameters, we have, for completeness, included in this scalar parameters section instances of the more common parameter types already described in previous section of this *Programmer's Guide*.

### REAL/float/NUMBER Parameters

The analyst uses this type of parameter when needing to supply a real value, e.g. some program guarantee expressed in dollars and cents. The `Mpu.h` definition will use the `NUMBER` specification. In the `pmaddent` call, the key argument is the `C_NUM` entry for `C_Type`. An appropriate template is --

```
pmaddent(pcp, "STDFA", (char *)&MP.STDFA, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

### INTEGER/int Parameters

The analyst uses this type of parameter when needing to supply a value that is inherently an integer, e.g. the typical number of weeks in the waiting period for unemployment insurance. The `Mpu.h` definition will use the `int` specification. In the `pmaddent` call, the key argument is the `C_INT` entry for `C_Type`. An appropriate template is --

```
pmaddent(pcp, "UIWAITWKS", (char *)&MP.UIWAITWKS, NULL, P_SCL, C_INT, 0, 0, NULL, 0);
```

### FLAG Parameters

The analyst uses this type of parameter when wishing to supply a "switch" value, e.g. an indicator that will specify whether certain other calculations are to be performed or not. The `Mpu.h` definition will use the `int` specification for such a parameter. In the `pmaddent` call, the key arguments are the `C_INT` entry for `C_Type` and the `E_FLAG` entry for `Edit`. An appropriate template is --

```
pmaddent(pcp, "FAFLAG", (char *)&MP.FAFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL, 0);
```

### FRACTION Parameters

The analyst uses this type of parameter when wishing to supply a value that is inherently a fraction, and thus more constrained in value than a float. Tax rates and contribution rates are good examples of this type of parameter. The `Mpu.h` definition will use the `NUMBER` specification for such a parameter. In the `pmaddent` call, the key arguments are the `C_NUM` entry for `C_Type` and the `F_FRACT` entry for `Format`. In the template call we suggest for this type of parameter, the user has chosen NOT to require an `Edit` check that will constrain the value between zero and unity; the template itself is --

```
pmaddent(pcp, "UIBASRATE", (char *)&MP.UIBASRATE, F_FRACT, P_SCL, C_NUM, 0, 0, NULL, 0);
```

### **OPTION Parameters**

The analyst uses this type of parameter when the parameter reflects a forced choice among a small fixed number of alternatives; a numerical value is used to indicate a nominal or qualitative selection. As an example of such a qualitative distinction, one might consider a parameter that indicates whether CPP/QPP deductions are to be treated as (1) a deduction in computing taxable income, or (2) a non-refundable credit in the calculation of taxes, or (3) a tax credit refundable at the federal income tax level, but not at the provincial income tax level. The `Mpu.h` definition for a FLAG parameter will use an `int` specification. In the `pmaddent` call, the key arguments are the `C_INT` entry for `C_TYPE`, the `E_OPT` entry for `Edit`, and the numeric entry giving the number of legitimate categories for the `Row-max` argument. An appropriate template is --

```
pmaddent(pcp, "MDCROPT", (char *)&MP.MDCROPT, NULL, P_SCL, C_INT, E_OPT, 2, NULL, 0);
```

### **EDIT-FRACTION Parameters**

The analyst uses this type of parameter when it is desirable to constrain any user-supplied value to fall in the interval from zero to unity. For example, the parameter might represent a taxback rate that would be considered unreasonable if it corresponded to a rate of less than zero percent or greater than one hundred percent. The `Mpu.h` definition for an editable fraction parameter will use a `NUMBER` specification. In the `pmaddent` call, the key arguments are the `C_NUM` entry for `C_Type` and the `E_FRCT` entry for `Edit`. The user might wish also to specify a `Format` specification of `F_FRACT`. An appropriate template is -

```
pmaddent(pcp, "CHATR1", (char *)&MP.CHATR1, NULL, P_SCL, C_NUM, E_FRCT, 0, NULL, 0);
```

### **DUMMY Parameters**

The user will not typically specify DUMMY parameters, which are intended for conveying labeling and sectioning information when parameter configurations are being documented. An illustrative template is -

```
pmaddent(pcp, "2.3.1", NULL, NULL, P_DUMMY, C_STR, 0, 0, NULL, 0);
```

For all types of scalar parameters, the user has the choice among mechanisms for supplying values to them:

1. specification via inclusion of the parameter in a parameter file (MPR, CPR and APR files),
2. specification via presence in a supplementary inclusion parameter file (MPI, CPI and API), and
3. specification via the SPSM's dynamic parameter editing facility. (Note, however that the ability to use the third option may be constrained by the parameter's `pmaddent` entry for the `Edit` argument.) This approach is automatic if the user chooses not to specify a value; the `Edit` argument permitting, the SPSM will prompt for a value.

## **VECTORS OF USER-DEFINED PARAMETERS**

The preceding portions of this chapter have focused primarily on scalar parameters, in part

because they are the most common types, and in part because they are easiest to describe. However, the SPSM also offers the user the capacity to create vectors of parameters. Such vectors will be most relevant when the user wants to create a set of related parameters with the members of the set occurring in a natural "indexable" order along a single dimension.

As an example, consider the case of an analyst modeling some proposed housing supplement program. For each family size up to ten this hypothetical program has an income limit beyond which a family becomes categorically ineligible to receive benefits. Unfortunately, these limits, though increasing with family size, are not related to that family size in any smooth or readily calculated manner. Instead, the user wants to have ten different parameters, corresponding to families of size one to ten-plus, to represent the benefit cutoff levels. It makes much more sense to have a vector of parameters, indexed on family size, than to develop code that treats each of the ten possibilities as a separate, independently developed case.

In this section then, we characterize the key points the user must understand to define vectors of user parameters for SPSM models. Our earlier comments about parameter addition in general continue to hold (order of changes to files, use of mnemonic values, validation, etc.), but we focus on those aspects specific to the effective use of vectors of user-defined parameters.

#### **Additions to `Mpu.h`, `Cpu.h` or `Apu.h`**

Just as the user declares scalar parameters in `Mpu.h` (or `Cpu.h` or `Apu.h`), s/he must also declare any user-defined parameter vectors in these files. The scalar and vector declarations look very similar, except that the vector declaration indicates, via an expression in square brackets, the length of the vector. The SPSM treats parameter vectors as column vectors; thus the length of the vector is its number of rows.

For our housing program example, suppose that the user has declared a (manifest) constant `HHPYCOMR` (Hypothetical Housing Program, Income Cutoff Maximum Rows). The user has assigned it the value 10 because there will be a distinct cutoff for each family size up to ten-plus. The definition would be accomplished via a statement of the form --

```
#define HHPYCOMR 10    /* maximum # of number of rows in the HHPYCO vector */
```

See the `Mp.h` file in the `SPSM\DEFS` subdirectory (starting at about line 40) for illustrations using parameter vectors that are part of the black box SPSM, rather than being user-defined.

The vector itself is to be named `HHPYCO`, with the value of the *i*'th entry corresponding to the cutoff for a family of size *i*+1. (Recall that the C language starts all vectors with the zero'th entry.) The `Mpu.h` entry for the new vector will then look something like --

```
NUMBER HHPYCO[HHPYCOMR]; /* Hypothetical Housing Program Income Cutoffs */
```

Although it is possible to "hardwire" the length directly into the declaration, e.g. using something like `HHPYCO[10]`, we strongly discourage it. We recommend instead the manifest constant approach described above. The reason behind this recommendation stems from the need, in the corresponding `Ampd.c`'s `pmaddent` invocation, of an entry for the maximum number of rows. Using a given manifest constant in both locations precludes the

possibility of a later revision leading to one value being used in `Mpu.h` while another is used in `Ampd.cpp`. If the user should create a discrepancy between the `Mpu.h` (or `Apu.h` or `Cpu.h`) and `Ampd.cpp` values, the errors that result could be infuriatingly difficult to track down.

Recall that the actual number of rows present in the (column) vector for a given SPSM execution may be different from (less than) the maximum number possible for that parameter. Thus, the user must also declare, in the same header file, a variable in which the SPSM will store the actual number of rows being used (a value that may vary from run to run of a given executable version of a glass box model). The user provides a variable for the SPSM to store the actual number of rows via an additional declaration in the header file. Following the SPSM convention that these length variables are named as the parameter name with a suffix of "rows", the `Mpu.h` file should also contain a declaration of the form --

```
int HPPYCOrows;          /*    number of rows in HPPYCO    */
```

The `mp.h` file in the `SPSM\DEFS` subdirectory provides many examples in its section on array limits (about line 580). Later on, `Ampd.cpp`'s `pmaddent` call for HPPYCO will refer to the address of the HPPYCOrows variable.

### **Additions to `Ampd.cpp`**

So that the SPSM can make the values in the new parameter vector available to the user's substantive code, the user must set up the appropriate linkages via an invocation of `pmaddent`, just as with scalar parameters. The invocation would look like one of the following:

```
pmaddent(pcp, "HPPYCO", (char *)MP.UM.HPPYCO, NULL, P_VCT, C_NUM, E_NONE,
HPPYCOMR, &MP.UM.HPPYCOrows, 0);
```

or

```
pmaddent(pcp, "HPPYCO", (char *)&MP.UM.HPPYCO[0], NULL, P_VCT, C_NUM, E_NONE,
HPPYCOMR, &MP.UM.HPPYCOrows, 0);
```

In the first illustrative invocation the third argument uses no ampersand because the reference is to the new parameter vector; C treats such a reference as the address of the first element. In the second illustrative invocation the user has elected to refer more explicitly to the address of the first element by including the ampersand and the [0] index. The `MpdX.cpp` files in the `SPSM\MODEL` subdirectory contain examples of both types of reference.

Three other `pmaddent` arguments deserve special comment for our description of the highlights for user-defined parameter vectors. The `Agg_Type` argument (#5) necessarily takes on the value `P_VCT`. The `Row-max` argument (#8) is the manifest constant created in `Mpu.h` to specify the maximum number of rows; in our housing program example this corresponds to the HPPYCOMR entry. Finally, the `Rows-addr` entry (#9) corresponds to the name of the variable declared to store the actual number of rows, preceded by an ampersand; in our housing program example this corresponds to the `&MP.UM.HPPYCOrows` entry.

Note that other capacities activated by `pmaddent`'s arguments remain available to the user. Thus, `C_Type` is used to indicate whether the variable is a float value or an integer. The user can use the `Format` argument to specify, if desired, a format for each of the individual values

in the vector. And the user employs the Edit argument to impose any relevant edit checks.

Just as with scalar parameters, the user will also wish to modify the `Ampd.cpp` file to add an invocation of `stradd` for each new user-defined parameter vector. This addition will ensure that when the SPSM documents the new user-defined parameter, the user's textual description of the parameter will form part of that documentation.

### User-Defined Parameter Vector References in the Source Code

Once the user has completed the header file and `Ampd.cpp` changes necessary to make the parameter vector available to the substantive functions, it remains to refer to the relevant parameter values in those substantive functions. To continue with the hypothetical housing program example, suppose that the user has available an integer variable, `HHPFS`, (Hypothetical Housing Program Family Size) that gives the family size as defined by the anticipated regulations governing the program. Suppose too, that the user is absolutely confident that `HHPFS`'s value will lie in the domain 1 through 9 inclusive. To refer to the relevant income cutoff for benefits from the hypothetical program, the user, recognizing that the C language always numbers a vector's elements starting with 0, would employ an expression of the following form:

```
MP.UM.HHPYCO[HHPFS-1]
```

### Specification of Parameter Vector Values

In order for the user's new code to accomplish anything, the values of the vector's elements must be made available to the SPSM so that it, in turn, can make them available to the user's code. Typically, the user will specify these values in an ".MPR" or ".MPI" file (or their ".CPR", ".CPI", ".APR" or ".API" counterparts). The `UIREPUER` vector, specifying key regional unemployment as they apply to UI entry requirements for repeaters, provides a good example.

```
UIREPUER      5          # Regional unemployment rate
    6.0
    7.0
    8.0
    9.0
   11.5
```

The format is clear. The first line contains the name of the parameter, followed by the number of ACTUAL elements to be used; an optional documentary comment should be added to make the nature of the parameter obvious to any reader of the file. Successive lines specify, one value per line, the values for the vector. It is important that the number of elements entry not exceed the maximum rows value specified in the `pmaddent` entry, and that the number of additional lines in the parameter file be equal to the number on the parameters first line; the SPSM will check to ensure that these requirements are met.

To continue with our hypothetical housing program example, the user might enter, in the ".MPR" or ".MPI" file, something like the following:

```
HHPYCO      10  # Income cutoffs for housing program, by family size
    5000.0
    6120.0
    7250.0
    8400.0
```

```
9500.0
10600.0
11600.0
12500.0
13300.0
13900.0
```

## Summary

The key factors in adding vectors of user parameters to an SPSM glass box model can be summarized in the following checklist:

1. Make appropriate changes in the header file (e.g. `Mpu.h`).
  - Use a manifest constant for the maximum length of the vector, e.g.
  - `#define HHPYCOMR 10 /* maximum # of rows for HHPYCO */`
  - Declare the vector itself,
  - `NUMBER HHPYCO[HHPYCOMR]; /* comment */`
  - Declare a variable to hold the actual length of the vector, e.g.
  - `int HPPYCOrows; /* actual number of rows in HPPYCO */`
2. Make appropriate changes in the `Ampd.cpp` file; remember the benefits of partial compilation.
  - Insert an appropriate `pmaddent` invocation, usually by modifying a copy of an existing one.
  - Enter an invocation of `stradd` so that the SPSM can label the new parameters when appropriate.
3. Write the C-language source code that uses the parameters. Remember C's convention that vectors begin with the zero elements. Debugging compilation is often useful here too.
4. Supply values for the elements of the vector via a multi-line entry in an appropriate parameter file.
5. Don't forget the need for validation and testing to make sure that the new code is doing what is intended of it.

## USER-DEFINED SCHEDULES FOR LOOKUPS

Parameters in the form of schedules are useful primarily when one needs to perform some sort of a lookup, i.e. given a x-value, find the corresponding y-value. This section employs as examples two schedules already present in the SPSM, and one hypothetical new user-defined schedule to be added as a parameter. Together, the three examples cover the major forms of schedule parameters that a glass box user might normally need.

The first of the existing schedule examples involves federal taxes -- given taxable income, calculate the corresponding tax from the tax table/schedule.

The second existing schedule example addresses program take-up rates -- assuming that the decision of whether to apply for benefits in a program is believed to depend on the benefit

that could be claimed (the higher the benefit that would be received, the more likely a unit is to file to claim that benefit), given a unit's potential benefit, look up its probability of applying for (taking up) those benefits.

The third, new parameter, example involves a totally hypothetical earnings supplement based very loosely on the U.S. Earned Income Tax Credit, but applied to individual earnings. In it, a hypothetical earnings supplementation program subsidizes initial earnings, up to \$10,000 annually, at a rate of 15%, does not further subsidize any earnings from \$10,000 to \$15,000, and then, beyond \$15,000, reduces the subsidy previously given at the rate of 10% of earnings above \$15,000, so that there is no subsidy payable to individuals earning \$30,000 or more. The new parameter will describe the subsidy payable as a function of the individual's earnings. The relevant coordinate pairs are thus ( 0, 0 ), ( 10000, 1500 ), ( 15000, 1500 ), and ( 30000, 0 ).

In terms of their specification as SPSM parameters, schedules are very similar to vectors. The main exception is that schedules have a fixed number of columns, three, rather than the single column for a vector. (In use, the schedules employ the SPSM's lkup1 and lkup2 functions.) Thus, with the relatively minor exceptions highlighted in this section, one adds a schedule to a glass box application very much as one would add a vector of parameters. Consequently, the vector-oriented prescriptions about mnemonic names, stradd labeling, partial compilation, validation etc. are not repeated here.

### **Schedule Types and Lookup Functions**

An appreciation of two separate dichotomies is absolutely critical for the effective use of schedules in the SPSM.

The first dichotomy involves the type of schedule. The user makes the choice as to type via the fifth argument of the pmaddent call.

If the argument is P\_LKPXY, then lookups in the schedule are done in X-Y format, using the first (x-values) column of the schedule and the second (y-values) column; the slope values of the third column (the slopes across the successive segments of the schedule) are present, but ignored (that information being redundant because it could be calculated from the X-Y pairs). If the fifth pmaddent argument is P\_LKPSL, then lookups in the schedule are done in slope format, using the information in the first (x-values) column and the third (slopes) column, plus the first value in the second (y-values) column. The remaining values in the second column are ignored in the sense that they are redundant because they could be calculated using the rest of the information in the schedule.

The second dichotomy reflects whether or not the user wishes to apply interpolation in the calculation when performing the associated lookup with the schedule. When interpolation is desired (when the desired value might lie BETWEEN entries in the y-values column), the user invokes the lkup1 function from the SPSM algorithm library. When no interpolation is desired, the user invokes the lkup2 sister function. The [Algorithm Guide](#) provides the authoritative description of these two algorithms.

### **Appearance in SPSM Header Files**

Exactly as with vectors of parameters, user-defined parameters that are schedules require certain entries in an appropriate header file (`Mpu.h`, `Cpu.h`, or `Apu.h`).

One of these is (usually) a manifest constant to define the maximum length of the schedule. The federal tax schedule (FTX) uses the maximum length `FTXMAX`. The GIS single pensioner take-up schedule (GISST) uses `GISSTMAX`. For our earnings supplement schedule, ESS, we'll use `ESSMAX`. The corresponding definitions (in `Mp.h` for `FTXMAX` and `GISSTMAX`, and in `Mpu.h` for `ESSMAX`) are as follows:

```
#define FTXMAX    15      /* maximum of number of rows in FTX table      */
#define GISSTMAX  8       /* maximum of number of elements in GISST table */
```

and

```
#define ESSMAX    5       /* maximum number of rows in ESS schedule      */
```

The second of these is a variable in which the SPSM stores the actual number of rows used by the schedule in a given run; it must, of course, be less than or equal to the maximum number. Following SPSM conventions, the `Mp.h` definitions for variables to contain the actual numbers of elements are as follows:

```
int  GISSTrows;          /* number of rows in GISST table */
int  FTXrows;            /* number of rows in FTX         */
```

In `mpu.h`, we'll follow this convention and define a variable `ESSrows` for the actual number of rows in ESS --

```
int  ESSrows;           /* number of rows in ESS schedule */
```

`Mp.h` (for the FTX and GISST schedules) and `Mpu.h` (for the ESS schedule) also need to contain the definitions for the schedules proper. Typically, these are carried out using the manifest constants defined earlier. The SPSM provides a constant, `LKP_COLS`, that indicates clearly its role as defining the number of columns for lookup schedules. The definitions themselves are straightforward:

```
NUMBER FTX[FTXMAX][LKP_COLS]; /* Federal tax table [taxable income,basic federal
tax] */
NUMBER GISST[GISSTMAX][LKP_COLS]; /* GIS take-up rate: single pensioner by benefit
level [benefit,rate] */
NUMBER ESS[ESSMAX][LKP_COLS]; /* Earnings supplement schedule [earnings, benefit
level] */
```

### Appearance in `pmaddent` Calls in `Ampd.c`

The user defining schedule parameters will need to modify the `Ampd.cpp` file, adding invocations of `pmaddent`, to enable the SPSM to make the parameter available to the substantive source code. We begin by looking at the relevant `pmaddent` entries for the SPSM's existing FTX and GISST schedules.

The FTX example, drawn from the `Mpd2.cpp` file, appears as follows:

```
pmaddent(pcp, "FTX", (char *)&MP.FTX[0][0], NULL, P_LKPSL, C_NUM, 0,
FTXMAX, &MP.FTXrows, 0);
```

Note that the third argument indicates clearly that the schedule has both rows and columns, and that the fifth argument denotes this as a slope-oriented schedule; the eighth and ninth arguments make use of the manifest constant and actual-number-of-rows entries defined in



`mp.h`.

The GISST example, drawn from the `Mpd1.cpp` file, appears as follows:

```
pmaddent(pcp, "GISST", (char *)&MP.GISST[0][0], F_LKTUR, P_LKPXY, C_NUM, E_FRCT,
GISSTMAX, &MP.GISSTrows, 0);
```

Here the fifth argument indicates that this is an X-Y type schedule. Again, the eighth and ninth arguments make use of the elements defined for the schedule in the `mp.h` file.

For the hypothetical earnings supplementation program, we would add to the `Ampd.cpp` file an invocation of `pmaddent` (probably copied from an existing call and then modified as appropriate) that appears as follows:

```
pmaddent(pcp, "ESS", (char *)&MP.UM.ESS[0][0], NULL, P_LKPXY, C_NUM, 0, ESSMAX,
&MP.UM.ESSrows, 0);
```

The strong parallels with the existing GISST schedule should be apparent. Note, however, the key differences that mark a user-defined parameter schedule: the UM qualifier in the third and ninth arguments, and the user-defined (maximum rows) constant and (actual rows) variable address for the eighth and ninth `pmaddent` arguments.

### Employing Schedule References in User Code

Glass box applications that use schedules will reference them almost exclusively via the SPSM's two lookup functions, `lkup1` and `lkup2`. This makes source code expressions using the parameters very straightforward. Illustrations using our three examples indicate the nature of these references.

The GLASS subdirectory's `ATXCALC.CPP` function serves to calculate federal income taxes. This computation involves looking-up, for an individual, that individual's tax as a function of his/her taxable income. The user chooses whether or not to apply interpolation (via the choice between `lkup1` and `lkup2`), supplies the schedule, the actual number of rows, and the relevant x-value, and the lookup function does all the rest automatically. Here, the user does want interpolation, applied in a schedule. The relevant source code appears as follows:

```
if (isnzero(in->im.imitax)) {
/* calculate federal tax */
in->im.imfedtax = (NUMBER) lkup1(MP.FTX, MP.FTXrows, in->im.imitax);
DEBUG2("%s fedtax =%.2f\n", in->im.imfedtax);
}
```

The GLASS subdirectory's `AGIS.CPP` function calculates GIS benefits. This computation involves looking-up, as a function of the potential benefit that would be payable, the probability that the unit will take-up (i.e. apply for) the benefit. Here the user chooses not to invoke interpolation -- the desired takeup rate is the one in the last row in which the potential benefit is at least as great as the row's x-value. The user provides the schedule, the actual number of rows, and the potential GIS benefit, and the lookup function returns the takeup probability. (Once again, the schedule itself appears in the next sub-section, ) The expression to ascertain the takeup probability appears as

```
lkup2(MP.GISST, MP.GISSTrows, (double) gis))
```

For the earnings supplement illustration, assume that the user has assigned the appropriate definition of earnings for an individual to a (double) variable named `iearn`. Then the expression for looking up the individual's corresponding earnings supplement would be --

```
lkup1(MP.UM.ESS, MP.UM.ESSrows, iearn)
```

Note the necessity for the UM qualifier indicating that ESS is a user-defined schedule.

### Appearance in Parameter Files

As with any other parameter, the user is responsible for defining schedule parameters in the appropriate parameter file (`.MPR/I`, `.CPR/I`, or `.APR/I`). In parallel with the specification of a parameter vector, the first line provides the parameter name and number of rows, along with a comment identifying the parameter. The remaining rows for the schedule are the x-value, y-value, slope triplets. Probably the only non-obvious characteristic is that the redundant items (those that will not be used for the computations) are enclosed in parentheses.

The slope-oriented FTX schedule describes tax payable (before tax reform) as a function of taxable income --

```
FTX      10          # Federal tax
              table
          0          0          0.060
        1238      ( 74)      0.160
        2476     (272)      0.170
        4952     (693)      0.180
        7428    (1139)      0.190
       12380    (2080)      0.200
       17332   (3070)      0.230
       22284   (4209)      0.250
       34664   (7304)      0.300
       59424  (14732)      0.340
```

The X-Y type GISST schedule describes take-up probabilities as a function of amount of GIS benefit available. The use of the `lkup2` function with this schedule means that these take-up rates are modeled as jumping sharply at the key benefit levels.

```
GISST    5          # GIS take-up rate: single pensioner by benefit
              level
          0          0.365          (0.0009)
         169          0.510          (0.0006)
         419          0.660          (0.0003)
         919          0.820          (0.0001)
        3169         1.000          (0.0001)
```

The X-Y type ESS schedule describes the earnings supplement benefit as a function of an individual's earnings; it is used with the `lkup1` function because interpolation is desired.

```
ESS      4          # Hypothetical earnings supplement schedule
          0          0          (0.15)
```

10000	1500	( 0.00 )
15000	1500	( -0.10 )
30000	0	( 0.00 )

### Key Points for Adding Schedule Parameters

Most of the key points for schedule parameters are identical to those for vector parameters.

1. Modify the relevant header file to include a manifest constant for the maximum number of rows, an integer variable to store the actual number of rows, and the definition for the schedule itself.
2. Modify the `Ampd.cpp` file to include appropriate `pmaddent` and `stradd` invocations, generally ones copied from elsewhere and then modified.
3. Provide the schedule via an appropriate parameter file or parameter inclusion file, and don't forget to validate the addition.

Two other key points are specific to schedule parameters.

1. Be absolutely sure, in the parameter file, that the x-value column of the schedule contains values that are in strictly ascending order.
2. Don't forget to "mark" the redundant values in the schedule by enclosing them in parentheses.

### ADDING MATRICES OF PARAMETERS

For some specialized purposes involving groups of parameters, even vectors or schedules of parameters are not sufficiently convenient. For example, rather than managing several equal-length vectors in parallel, it may be much more efficient to perform lookups in a matrix of values. The design of the SPSM permits the definition and utilization of such matrices, though it limits the number of dimensions to 2 (rows and columns). This section will describe the use of matrices of parameters via two examples, one drawn from the black box version of the SPSM, and a second involving the specification of a new user-defined matrix of parameters. Given the close relationship between parameter vectors and parameter matrices, there is no special highlights division for this section.

The black box illustration uses the CTPRST matrix specific to the commodity tax capacities of the SPSM. This parameter provides a large (48 commodities (rows) by 10 provinces (columns)) matrix of factors relevant for the calculation of the provincial sales tax.

The second example, in which the user adds a new matrix of parameters to the SPSM, involves a matrix of income cutoffs levels for an (hypothetical) experimental poverty measure. To facilitate the classification of families as in or out of poverty, the user wants to have a matrix that provides the relevant cutoffs as a function of integer variables specifying the families' structures (rows) and the sizes of place of residence (columns). Thus, the matrix's (3,2) entry will contain the poverty line for a family whose structure index is 3 and whose size of place of residence index is 2. The user has elected to name this matrix

EPMCO (experimental poverty measure cutoffs). For the sake of this example, we'll assume that the user has chosen a measure defined in terms of 18 family structures (involving, say, combinations of the numbers and ages of family members) and four categories of size of place of residence.

### Appearance in `Mpu.h`

Taking the black box matrix example first, we are not surprised to find the relevant header information for CTPRST in file `Mp.h` of the DEFS subdirectory. Thus, there is an integer definition, to define the actual number of rows (commodities) CTNUMCOM, as follows:

```
int CTNUMCOM;    /* number of rows for commodity dimension parms    */
```

In addition, there is a definition for the matrix itself --

```
NUMBER CTPRST[NUMCOM][NUMREG]; /* Provincial retail sales tax [com x prov] */
```

However, `Mp.h` does not contain manifest constants for the dimensions of the matrix (NUMCOM and NUMREG) since these are so closely related to the design of the commodity tax facility in the SPSM that they have been defined elsewhere so that the commodity tax module can more conveniently use the constants.

Turning to our experimental poverty measure cutoff matrix, we appreciate that we shall have to provide the relevant "defining" information to the SPSM via entries in the `Mpu.h` file. The specific needs are (1) manifest constants for the dimensions, (2) a variable for the actual number of rows, and (3) the matrix itself. The `Mpu.h` lines for these items might appear as follows:

```
#define EPMFAMMAX 18 /* maximum of number of family structures (rows) for EPMCO matrix */
```

```
#define EPMSIZE 4 /* number of size of place of residence categories for EPMCO matrix */
```

```
int EPMCOrows; /* number of rows for EPMCO matrix */
```

```
NUMBER EPMCO[EPMFAMMAX][EPMSIZMAX]; /* experimental poverty measure cutoffs [fam x size] */
```

### Appearance in `Ampd.c`

In parallel with the requirements for vectors of parameters, the SPSM requires for each parameter matrix a call to `pmaddent` so that the parameter values can be made available to the user's source code.

For our black box example, this call, found in file `Mpd4.cpp`, appears as follows: (There is, of course a corresponding `stradd` call.)

```
pmaddent(pcp, "CTPRST", (char *)MP.CTPRST, NULL, P_TBL, C_NUM, E_FIXL,
NUMCOM, &MP.CTNUMCOM, NUMREG);
```

The only arguments of any special interest at this point are the `P_TBL` entry for the fifth (Agg\_Type) argument, and the `NUMREG` entry for the final (number of columns) argument. The eighth and ninth entries (maximum and address of actual numbers of rows) are just as we would expect them given the preceding descriptions for vectors and schedules.

Turning to our poverty measure glass box example, we recognize that it is necessary to add a `pmaddent` call to the `Ampd.cpp` file to permit the SPSM to give the user's source code access to the parameter matrix. That call might well appear as follows:

```
pmaddent(pcp, "EPMCO", (char *)MP.UM.EPMCO, NULL, P_TBL, C_NUM, E_NONE, EPMFAMMAX, &MP.UM.EPMCOrows,
EPMsize);
```

Presumably, the user would also add to the `Ampd.cpp` file a call to `stradd` to permit the SPSM to produce appropriate documentary information.

## Referencing Matrix Elements in Source Code

Referencing the elements of a parameter matrix is easy. Assuming the variable `i` holds the (integer) commodity category and variable `j` the (integer) province code, then the associated removal factor for that combination is --

```
MP.CTPRST[i][j]
```

Similarly, if the integer variable `fstruct` holds the family structure code, and the integer variable `sizecode` provides the category for the size of place of residence, then the experimental poverty measure cutoff for that structure/size combination is given by --

```
MP.UM.EPMCO[fstruct][sizecode]
```

The primary factor to consider in such references is the C-language's convention that each dimension begins with the zero element; e.g. our 18 by 4 array uses indices that run from 0 through 17, and 0 through 3, respectively. A user must make the decision about the appropriate tradeoff between using "natural, positive" integers as indices into the matrices, and economizing on the fixed block of memory available for user parameters (including any necessary row address variables).

## Appearance in Parameter Files

Just as with all other forms of parameters, the user must provide values for the parameters. Normally this will occur via entries in the appropriate parameter or parameter inclusion files (i.e. `.MPR`, `.MPI`, `.CPR`, `.CPI`, `.APR` or `.API`). For parameter matrices a parameter file entry consists of a first line that specifies the name of the parameter and the actual number of rows, plus typically a documentary comment. The succeeding lines for the parameter then supply the rows of the matrix. In our illustrations here, we provide only the first, identifying, line and then the first of the lines of numeric values.

For the black box example --

```
CTPRST      40      # Provincial retail sales tax
0.01326 0.01326 0.01326 0.01326 0.01316 0.01406 0.02242 0.00626 0.00010 0.00550
0.15257 0.15257 0.15257 0.15257 0.13057 0.24354 0.15684 0.13914 0.00013 0.29100
0.17538 0.17538 0.17538 0.17538 0.16338 0.22635 0.13837 0.08953 0.00010 0.00605
0.08125 0.08125 0.08125 0.08125 0.08424 0.07750 0.06300 0.08521 0.00009 0.07406
0.08029 0.08029 0.08029 0.08029 0.07239 0.06953 0.05715 0.07306 0.00010 0.06512
0.08293 0.08293 0.08293 0.08293 0.06684 0.05282 0.05581 0.00305 0.00008 0.06866
0.00296 0.00296 0.00296 0.00296 0.00359 0.00197 0.00130 0.00171 0.00001 0.00141
0.00997 0.00997 0.00997 0.00997 0.00934 0.00753 0.01018 0.01073 0.00024 0.01057
0.00886 0.00886 0.00886 0.00886 0.01140 0.01421 0.00969 0.00879 0.00022 0.01017
0.08363 0.08363 0.08363 0.08363 0.06777 0.00206 0.02368 0.04331 0.00004 0.00662
0.08283 0.08283 0.08283 0.08283 0.35376 0.00201 0.02646 0.00544 0.00004 0.02263
0.09406 0.09406 0.09406 0.09406 0.06143 0.00733 0.01685 0.01645 0.00064 0.02582
0.08515 0.08515 0.08515 0.08515 0.07698 0.09175 0.07097 0.06762 0.00011 0.08368
```

```

0.08160 0.08160 0.08160 0.08160 0.09371 0.08702 0.06739 0.06646 0.00008 0.07739
0.08086 0.08086 0.08086 0.08086 0.08141 0.08654 0.06925 0.06538 0.00009 0.07740
0.08238 0.08238 0.08238 0.08238 0.08320 0.08203 0.06751 0.05395 0.00011 0.07746
0.08331 0.08331 0.08331 0.08331 0.09420 0.01711 0.07477 0.01461 0.00009 0.01935
0.00067 0.00067 0.00067 0.00067 0.00054 0.00464 0.00740 0.00678 0.00006 0.00690
0.05967 0.05967 0.05967 0.05967 0.05408 0.04822 0.02270 0.01925 0.00017 0.01865
0.00821 0.00821 0.00821 0.00821 0.01031 0.00618 0.00623 0.00397 0.00011 0.00738
0.00043 0.00043 0.00043 0.00043 0.00034 0.00124 0.00145 0.00173 0.00002 0.00059
0.01581 0.01581 0.01581 0.01581 0.00875 0.10256 0.01323 0.00799 0.00025 0.01145
0.02112 0.02112 0.02112 0.02112 0.02389 0.04246 0.03516 0.00786 0.00013 0.01465
0.07207 0.07207 0.07207 0.07207 0.06970 0.08270 0.07019 0.04924 0.00005 0.10050
0.07667 0.07667 0.07667 0.07667 0.07584 0.08081 0.06841 0.03319 0.00014 0.04053
0.14145 0.14145 0.14145 0.14145 0.14506 0.01002 0.00841 0.00897 0.00012 0.01248
0.04574 0.04574 0.04574 0.04574 0.04843 0.08112 0.03185 0.02851 0.00021 0.02790
0.03739 0.03739 0.03739 0.03739 0.04921 0.01000 0.02035 0.01185 0.00019 0.01653
0.08336 0.08336 0.08336 0.08336 0.08897 0.07353 0.06346 0.06354 0.00003 0.04449
0.07581 0.07581 0.07581 0.07581 0.08182 0.07966 0.05424 0.06289 0.00007 0.07054
0.07746 0.07746 0.07746 0.07746 0.08965 0.04561 0.05949 0.03563 0.00009 0.04247
0.04765 0.04765 0.04765 0.04765 0.04967 0.02692 0.02058 0.02111 0.00016 0.01419
0.00489 0.00489 0.00489 0.00489 0.00411 0.00745 0.00795 0.00733 0.00017 0.00929
0.08402 0.08402 0.08402 0.08402 0.11465 0.08444 0.06428 0.06551 0.00008 0.07433
0.07875 0.07875 0.07875 0.07875 0.07826 0.08018 0.07052 0.06623 0.00015 0.07777
0.04826 0.04826 0.04826 0.04826 0.04245 0.00867 0.00918 0.00758 0.00008 0.01028
0.06598 0.06598 0.06598 0.06598 0.07010 0.05898 0.07703 0.01556 0.00707 0.02343
0.02430 0.02430 0.02430 0.02430 0.02547 0.02539 0.00705 0.00708 0.00018 0.01004
0.01002 0.01002 0.01002 0.01002 0.01255 0.00805 0.00822 0.00735 0.00029 0.01300
0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000

```

For our postulated poverty measure example --

```

EPMCO      18      # Experimental poverty measure cutoffs
              5600.0
              6210.0
              6530.0
              7050.0

```

## SUMMARY/CONCLUSION

It is useful to conclude by highlighting, but without redeveloping them in any detail, the general level key points relevant for adding less typical scalar parameters and non-scalar parameters to a model. In noting these points, we are assuming that the analyst is following the general procedures outlined for scalar parameters. For example, it is taken as given that the analyst is working with COPIES of all the relevant files, and is performing all of the modifications in a task subdirectory dedicated to the analysis at hand. We also assume that the user has updated the project environment, and is following the appropriate "checklists" provided for the less typical parameters.

1. We recommend the "copycat" approach as general way to proceed. Throughout this chapter we've provided concrete illustrations of the items a user might want to use as templates. Users should rarely need to employ the detailed material on `Mpu.h/Cpu.h/Apu.h` (definitions, manifest constants for max rows, and actual rows) and `Ampd.cpp` (`pmaddent` and `stradd`).
2. Advanced users may want to be aware of the special "services" available via the `pmaddent` arguments: the ability to specify printing formats, edit checks, and the maximum number of allowable rows or options.
3. Vectors can sometimes be much more efficient than a number of individually named scalar parameters. The SPSM provides for this capacity, though the user must supply additional information in the `pmaddent` call and be sure to provide another variable for

the number of relevant rows, as well as a constant for dimensioning. We've offered several potential templates to facilitate the copycat approach.

4. In many respects, schedules are like a special case of vectors, applicable when one needs to look up a y-value, as a function of a x-value, from a fixed relationship.
5. Matrices (2-dimensional) are also possible. Some additional information, the number of columns, becomes necessary, but the matrix approach can be considerably more efficient than juggling multiple parallel vectors. Once again, the copy and modify approach is recommended.

## Glass Box Development: Adding New Variables

This chapter describes how to add new user-defined dependent variables to an SPSM glass box application. Thus, it shows how to address challenges such as those raised in the Quick Start example, where the user would like to have had a separate variable for the hypothetical Family Allowance supplement. The availability of user-defined dependent variables is even more important if the user is modeling some new program, e.g. an earnings supplement that could not be conveniently combined into any existing model dependent variable.

Structurally, this chapter covers all of the major issues and steps involved in adding new dependent variables to a model. Included in this chapter is an introductory overview of the process, and also a section that characterizes the major types of variables that the user may wish to add. An explanation follows of the critical `vardef` function that establishes the linkages between the user's source code and the rest of the SPSM and also describes how to use the `stradd` function to make the new variables' labeling available throughout the SPSM. We then introduce an illustrative extension of the Family Allowance supplement example used in previous examples, which defines new variables that will be available to the SPSM's several output facilities. Following this are examples of the source code changes that the user has to make, and the descriptions of the compilation and validation of the resulting model.

### OVERVIEW FOR ADDING VARIABLES

In broadest outline, the key steps involved in adding new variables can be characterized as follows.

1. Decide what new dependent variables are needed, choose appropriate names and descriptions for them, and copy all of the relevant header and source code files over to the subdirectory in which the new model will be built.
2. Make the relevant changes to the project environment (identifying all of the appropriate source code files associated with the new dependent variables), and update `Adrv.cpp` (providing documentary text strings).
3. Make the necessary changes to `vsu.h` and `vsdu.cpp` to render the new dependent variables accessible throughout the SPSM model that will be created.

4. Supply new source code (in new or existing modules) to calculate the values for the new dependent variables.
5. Compile the new model and validate it for correctness.

The preceding points are, of course, only an overview. Section on adding parameters and the recapitulation section provide a much fuller description of the model creation process as a whole. This chapter, however, concentrates on those details especially relevant for the addition of new dependent variables.

## **DEPENDENT VARIABLE TYPES AND CHARACTERISTICS**

The SPSM provides users with the capacity to create three different types of user-defined dependent variables. All three types are scalars. The SPSM does not provide for vectors or matrices of dependent variables. The specific types are as follows:

1. Numeric analysis -- This is the most common type of user-defined dependent variable. It consists of a numeric (float) value that will be used as an analysis variable, e.g. tabulated as a cell entry in the control parameter XTSPEC. A good example of this type of dependent variable is the value of some new income-tested benefit that will be payable to a family.
2. Integer analysis -- Less frequently used, this type of dependent variable consists of an integer (int) value that will be used as an analysis variable. The primary use of this type of variable is export in SAS format, where an integer variable takes up fewer characters than a numeric analysis variable. Examples of this type of variable might be the minimum and maximum numbers of weeks that a family could be without earnings income during the year (as deduced from the labour-force variables for the family members, e.g. weeks without work and looking for work).
3. Integer class -- this type of dependent variable consists of an integer (int) value that will be used as a classificatory variable, e.g. to define the categories for a classificatory variable in the XTSPEC parameter. This type of variable is particularly relevant when its values represent purely nominal categories, e.g. a classification of families by types.

A few other characteristics of user-defined dependent variables, individually and collectively, will be of considerable importance to the glass box user --

First, all user-defined dependent variables are defined at the level of the individual. Thus, the user must take care to assign values to "appropriate" individuals so that when the unit of analysis is at a higher level, say the census family level, the SPSM's roll-up algorithms will yield the desired results.

Second, the space allocated for such variables can handle approximately 50 variables. Violation of this limit can result in obscure errors that are difficult to track down.

## **THE VARDEF AND STRADD FUNCTIONS AND THEIR ARGUMENTS**

The vardef and stradd functions are absolutely critical to the capacity to create new user-



defined variables and have them used properly throughout the rest of the SPSM. It is only via the information communicated via calls to these functions that the rest of the SPSM learns about the nature of the new variables and the documentary text that goes with them. This section documents first the vardef function, and then the stradd function.

The vardef function plays the same general role for user-defined variables that pmaddent does for user-defined parameters. There will be one vardef call for each variable that the user defines. Vardef defines the characteristics of the new variable so that the SPSM can link it into the same variables framework used by the SPSD/M's own database, analytic and classificatory variables. The vardef calls are always made in the vsdu.cpp function. The following short description of the function's arguments appears at about line 100 of that function --

```
*   vardef("_uvew",      <= the name of the variable, quoted, with '_'
*       IN,              <= home structure (leave at 'IN')
*       im.uv.ew,        <= variable location (always in im.uv)
*       C_INT,           <= C-type (C_INT or C_NUM)
*       V_CLAS           <= type of variable (V_CLAS or V_ANAL)
*       );
```

We'll describe the nature of the vardef arguments one at a time, in order. Subsequent sections in this chapter provide specific illustrations for the use of both the vardef and stradd functions.

### **Vardef "Name" Argument (and Definition of Variable "Stem" Name):**

The first argument gives the variable's name as a double-quoted text string. The user should always include an underscore as the first character after the initial double quote, and then the characters "uv" as the second and third characters to indicate the "user variable" status. The remainder of the name, i.e. everything after the "\_uv" prefix, is known as the variable's stem name. Generally speaking, this stem portion should be as informative and mnemonic as is feasible.

For variables that will not be exported outside the SPSM itself, there is no real limit on the number of characters in the stem name. However, for variables that are to be exported to other packages, certain limitations may apply. For example, if the created variable is to be exported to SAS, then the stem must not exceed six characters. If it is to be exported to the MAPSIT EXAMINE module, then the stem portion should not exceed ten characters.

### **Vardef "Home Structure" Argument:**

The second argument indicates the structure in which the new variable resides. Because user-defined variables are ALWAYS defined at the individual level, the user should always enter this argument as an (unquoted) 'IN'.

### **Vardef "Variable Location" Argument:**

The third argument indicates the location of the variable (as regards the SPSM's data structures). The location is specified via three components, two of which are invariant. Specifically, the first portion of the location is ALWAYS equal to "im.uv" (but unquoted). This information tells the SPSM that the new variable is inside the user variable (uv) portion of the im structure (individual level model variables). The final portion of the location

specification is the new variable's stem name, as defined above for the first argument.

#### **Vardef "C-Type" Argument (C\_NUM & C\_INT):**

The fourth argument specifies the C language type of the variable. It will take on one of two values. Numerical analysis variables will use the entry "C\_NUM" (unquoted). Integer analysis and integer classification variables will use the value "C\_INT" (unquoted).

#### **Vardef "Usage" (Type) Argument (V\_ANAL & V\_CLAS):**

The fifth and last argument specifies whether the SPSM is to treat the variable as an analysis variable (tabulatable) or a classificatory variable (categorical). It will take on one of two values. Both numerical and integer analysis variables will use the entry "V\_ANAL" (unquoted). Integer classification variables will use the "V\_CLAS" (unquoted).

The combination of the fourth and fifth entries tells the SPSM how many bytes of memory it needs to allocate for the variables, an important consideration given the limit of 200 bytes for all user-defined variables. As noted above, the requirements are six bytes for a numerical analysis variable, three bytes for an integer analysis variable, and one byte for an integer classification variable.

We have already seen simple applications of the stradd function when we discussed the documentation of user parameters. The same function serves a similar purpose here, but in a more sophisticated fashion, since it is used to define both a short description of the user variables themselves, but also, in the special case of integer analysis and integer classification variables, the range of values and the textual labels associated with particular values of the variables. The vsdu.cpp file contains, at about line 110, capsule documentation for both the variable description and value label uses.

```
*      stradd("uvew",          <= the name of the variable, quoted
*          "Region"          <= a printing label for the variable
*          );
**      stradd("ew",          <= the stem name of the variable, quoted
*          "\tEast\tWest" <= string containing a label for each valid
*          );                  level, preceded by a tab '\t' character.
```

As with the vardef function above, we shall take up the arguments in sequence. A complicating factor here is that the NUMBER OF stradd INVOCATIONS and structure of the stradd arguments depends on the type of variable for which stradd is being used. However, the number of stradd arguments is always constant at two. Favoring clarity over brevity, we shall describe each of the three types (numeric analysis, integer analysis, and integer classification) individually.

#### **Stradd Calls for Numeric Analysis Variables:**

Numerical analysis variables require only a single invocation of the stradd function. The first argument specifies the variable name. It is identical to that used for the first vardef argument, EXCEPT THAT THE LEADING UNDERSCORE PRESENT THERE IS OMITTED HERE.

The second argument for a numerical analysis variable is the (quoted) string that the SPSM will use when it needs to print a description of the variable.

For example --

```
stradd("uvnewben", "New Hypothetical Benefit");
```

### **Stradd Calls for Integer Analysis Variables:**

Integer analysis variable additions require two separate stradd invocations. The first invocation defines the label for the variable as a whole. The second invocation defines, via a set of labels for the individual integer values, the range of values for the variable.

In the first (variable label) invocation, the first argument specifies the variable name. It is identical to that used for the first vardef argument, EXCEPT THAT THE LEADING UNDERSCORE PRESENT THERE IS OMITTED HERE.

In the first (variable label) invocation, the second argument is the (quoted) string that the SPSM will use when it requires a description of the variable as a whole, e.g. in documenting a table.

In the second (value labels) invocation, the first argument is the STEM NAME for the variable; neither the underscore nor the leading "uv" string should be present.

In the second (value labels) invocation, the second argument is a quoted string that tells the SPSM how many categories are relevant. The string consists of the repeated pattern 'tx' where x always varies from zero to "one minus the total number of categories". Thus, for a variable having four categories, the second argument would take the form -- "t0\t1\t2\t3". The slash-t notation is the C language's standard way of denoting a tab character.

For example --

```
stradd("uvnputpp", "Number persons unemployed 2+ periods");  
stradd("nputpp", "\t0\t1\t2\t3\t4");
```

### **Stradd Calls for Integer Classification Variables:**

The stradd invocations for integer classification variables are identical to those for integer analysis variables WITH ONE CRITICAL EXCEPTION. In the second (value labels) invocation, the second argument is a quoted string that provides the textual labels for the several categories of the variable. In essence, the several, user-supplied, labels correspond to the integers 0 .. "categories minus 1" entries of the second stradd invocation for an integer analysis variable. Thus, for example, the labels for "region" documentation might look as follows:

```
\tAtlantic\tQuebec\tOntario\tPrairies\tBritish Columbia
```

These labels, which may contain embedded blanks (since the tab characters serve as delimiters) would appear as labels when the user employed the SPSM's crosstabulation capacity or exported the new variable to a SAS file.

For example --

```
stradd("uvfamcat", "Nominal Family Income Category");  
stradd("famcat", "\tVery Poor\tPoor\tNear Poor\tNon-Poor\tRich");
```

Beyond the descriptive definitions of vardef and stradd arguments, appearing about lines 100-115 of the vsdu.cpp function, vsdu.cpp also contains template combinations of the

vardef and stradd calls for all three types of new variables. In typical SPSM fashion, users will normally find it convenient to modify copies of these templates when defining new variables. These templates appear at about lines 125-145 of vsdu.cpp.

```
* -----
*  A numeric variable:
*  -----
vardef("_xxxxxxx", IN, im.uv.xxxxxxxx, C_NUM, V_ANAL);
stradd("xxxxxxx", "Variable label");

* -----
*  An integer analysis variable, with values 0 through 4:
*  -----
vardef("_yyxxxxxx", IN, im.uv.yyxxxxxx, C_INT, V_ANAL);
stradd("yyxxxxxx", "Variable label");
stradd("xxxxxx", "\t0\t1\t2\t3\t4");
* -----
*  An integer class variable, with values 0 through 4:
*  -----
vardef("_yyxxxxxx", IN, im.uv.yyxxxxxx, C_INT, V_CLAS);
stradd("yyxxxxxx", "Variable label");
stradd("xxxxxx", "\tLABEL0\tLABEL1\tLABEL2\tLABEL3\tLABEL4");
```

## **THE FAMILY ALLOWANCE SUPPLEMENT EXAMPLE EXTENDED**

Although the preceding characterization of adding user-defined variables is complete from a definitional perspective, it is useful to see how the several steps look in practice. In this section we summarize the concrete example that the remaining sections will flesh out. In essence the example is a further extension of the Family Allowance exploration introduced in Quick Start and subsequently enhanced with the addition of user-defined parameters.

Our explicit objective here is to provide a worked example that gives concrete illustrations of all three types of user-defined variables, and to do so without burdening the reader with the overhead that would inevitably be associated with a completely new example. In the service of this objective we have not hesitated to sacrifice some realism (as to institutional motivation and practice) in favor of a clean, specific example.

We extend the Family Allowance supplement example by adding the following three user-defined variables:

1. A numeric analysis variable: the new variable is the gross amount of additional Family Allowance benefit received; we'll name it "uvfasup" (user variable, Family Allowance supplement). We shall assign this variable to the parent who reports the Family Allowance benefit for tax purposes.
2. An integer analysis variable: the new variable is the number of children in respect of whom the supplementary benefit is payable. We'll name the variable "uvncfasup" (user variable, number of children for Family Allowance supplement). We shall also assign this variable to the person reporting the FA for tax purposes. This type of variable finds a major use when exported in the SAS format because it takes up less space than a numeric analysis variable. The variable would also be useful as a tabulated variable to count the numbers of these children.
3. An integer class variable: the new variable categorizes the family by the number of

children in respect of whom the supplement is payable; we'll name the variable "uvfelfasup" (user variable, family classification for Family Allowance supplement). We shall use it primarily as a categorical variable for tables designed to validate our extensions to the FA supplement code. We shall assign this variable to the nominal head of the family. Note that this class variable is very similar to the integer analysis variable, but can be used directly as a row or column variable in a crosstabulation, whereas the integer analysis variable could not.

As we proceed to the actual changes and coding needed to implement these new user-defined variables, we assume that the relevant files (Adrv.cpp, vsu.h, vsdu.cpp, Afamod.cpp, SPSMGL.dsw, etc) have been COPIED over to an appropriate new subdirectory; here we'll assume that it is named GLASSEX3, this being our third worked glass box example.

### CHANGES TO PROJECT FILES AND ADRV.CPP

We begin by including all the relevant files into the project and by changing the name of the executable file in Project: Setting: Links to glassex3.exe.

The changes to adrv.cpp are simple, consisting entirely of (a) updating the short textual descriptions for the model and (b) indicating that Afamod (rather than famod) is to be used for Family Allowance calculations.

Of the two descriptions, the SPSM displays the first on its opening screen, to tell the user about the nature of the alternative system. The SPSM outputs the second description as part of the '.CPR' (control parameter) documentation that it produces when it runs the model. Recall that the positioning of this text (in the screen and in the output file) prevents the use of descriptions longer than 20 characters. After adding the new descriptions, the relevant portion of adrv.cpp (about line 35) appears as follows:

```
===== GLOBAL VARIABLE DEFINITIONS ===== */
/*global*/ char ALTNAME[IDSIZE+1] = "FA Suppl New Vars Ex";
/* Give global string describing version of this module */
/*global*/ char FAR Tdrv[] = "FA Suppl New Vars Ex"
#ifdef MSC
" [ " __TIMESTAMP__ " ] "
#endif
;
```

The altered line (about line 106) to indicate that the alternate driver uses Afamod.cpp, rather than famod.cpp, appears as --

```
Afamod(hh);          /* compute family allowances
```

Finally, compile a Debug version in Build:Start:Debug. The required links and compilations will be identified.

### CHANGES TO VSU.H

The file vsu.h serves to define the C language structure that holds the user-defined variables.

The relevant portion of this file, copied from the SPSM\GLASS subdirectory, appears as follows:

```
typedef struct uv_ {
    NUMBER    uvdummy;          /* dummy variable */
} uv_;
```

We replace the uvdummy line by three lines that define our new variables, uvifasup, uvncfasup & uvfclfasup. These new lines indicate the types of the new variables. After the changes, the new portion of vsu.h appears as follows:

```
typedef struct uv_ {
    NUMBER    uvfasup;          /* Family Allowance supplement payable */
    int        uvncfasup;       /* Number Children for FA supplement */
    int        uvfclfasup;      /* Family Class (Qualifying Children) for FA suppl */
} uv_;
```

Note the naming conventions used here. The typedef statement requires that the variables be prefaced with the uv prefix, but does NOT employ the leading underscore used in the vardef statements that appear later in the changes to vsdu.cpp.

One need not always modify, as we have done here, the GLASS version of vsu.h. If an already existing (user-defined) version of vsu.h contains user-defined variables that are to be retained, simply make a copy of that existing file and modify it as appropriate. Recall, however, that there is an overall limit of 200 bytes per individual for the user-defined variables.

## CHANGES TO vsdu.c

The necessary changes to the copy of vsdu.cpp consist of the vardef and stradd invocations that allow the SPSM to access the new variables and their documentation. Given the simplicity of these invocations, we use the example templates from the beginning of the file. We shall make these invocations as the end of the vsdu.cpp file, just before the final 'DEBUG\_OFF("vsdu");' statement. The additions appear as follows:

```
/* uvfasup: (Analysis) Family Allowance supplement payable */
vardef("_uvfasup", IN, im.uv.uvfasup, C_NUM, V_ANAL);
stradd("uvfasup", "Family Allowance Supplement");
/* uvncfasup: (Analysis) number of children for whom supplement paid */
vardef("_uvncfasup", IN, im.uv.uvncfasup, C_INT, V_ANAL);
stradd("uvncfasup", "# Children for FA Supplement");
stradd("ncfasup", "\t0\t1\t2\t3\t4\t5\t6\t7");
/* uvfclfasup: (Class) Family class by number of children for FA suppl. */
vardef("_uvfclfasup", IN, im.uv.uvfclfasup, C_INT, V_CLAS);
stradd("uvfclfasup", "Family Class for FA Supplement");
stradd("fclfasup", "\t0 Ch\t1 Ch\t2 Ch\t3 Ch\t4 Ch \t5 Ch\t6 Ch\t7 Ch");
```

Notice the second stradd call for each of the two integer variables, and the omission of the uv prefix in that (second) call that defines the number of cases (integer analysis variable) or the category labels (integer classification variable).

## CHANGES TO AFAMOD.CPP (OR, MORE GENERALLY, ANY NEW SUBSTANTIVE SOURCE CODE)

The preceding tasks have been preliminary to our central task, revision of Afamod.cpp to reflect the new calculation of Family Allowances, inclusive of the possible supplement to the

family. We are using `Afamod.cpp` here, but, more generally, at this stage, the user is ready to write/modify the source code necessary to make the desired changes to the calculation of SPSM variables, whatever modules those changes may involve. We'll illustrate the changes for our Family Allowances example one portion at a time, showing for each portion what the unmodified `Afamod.cpp` file looks like, and then how we have changed it to add our desired variables. References involving line numbers refer to the "original" version of `Afamod.cpp` found in the `SPSM\GLASS` subdirectory.

## Identifying String

Documentation is important. As we proceed through the `Afamod.cpp` file for our changes, we first update the description. Where the GLASS version of `Afamod.cpp` provides (at about line 39) the placeholder description --

```
/*global*/ char FAR Tfa[] = "Untitled"
```

we substitute a more informative description:

```
/*global*/ char FAR Tfa[] = "New Vars Version"
```

## Local Variables

Intermediate (local) variables can be very useful. Where the GLASS version of `Afamod.cpp` defines and initializes its local variables (about line 131), we add the new lines shown just below. The initialization of NUMBER/float variables with ZERO provides insurance against an obscure bug that shows up only on a few nonstandard machines.

```
/* user-defined intermediate (local) variables in support of glass box example 3
(user-defined SPSM variables) [using the "stem names" for two of the SPSM variables
being created] */
    NUMBER fasup = ZERO; /* amount of new FA supplement */
    int    ncfasup; /* number of children for whom supplement payable */
```

## Calculate and Assign the New Model Variables

We are now ready to calculate the new variables, and to assign them to the appropriate user-defined SPSM variables. For our `Afamod.cpp` example, we seek to calculate the amount of the possible supplement. We do so immediately after taxable and federal Family Allowances have been defined in the `SPSM\GLASS` version of `Afamod.cpp`, but before those values have been assigned as outputs from the `Afamod` routine. This condition occurs at about line 358. The relevant original source code appears as --

```
    else {
        DEBUG1("%s standard FA calculation\n");
        tfa = nch * MP.STDFA; /* taxable family allowances */
        ffa = tfa; /* federal part of family allowances*/
    }

    DEBUG3("%s tfa=%.2f, ffa=%.2f\n", tfa, ffa);
```

In the new code that we add, we are careful to make sure that an appropriate value is calculated for our intermediate variables, no matter what the nuclear family looks like, and that the taxable and federal Family Allowance variables are updated if the supplement is relevant. Notice that we are retaining the parametric structure developed in the section 6 of this guide.

```

/* Conditionally apply the Family Allowance bonus for the
 * "FASUPFECth" and subsequent children <18 in the unit,
 * including any necessary updates to taxable and federal FA */

if ((MP.UM.FASUPFLAG == 1) && (nch >= MP.UM.FASUPFEC)) {
    ncfasup = (nch-MP.UM.FASUPFEC+1);
    fasup = ncfasup * MP.UM.FASUPPC;
    tfa += fasup;
    ffa += fasup;
}
else {
    ncfasup = 0;
    fasup = ZERO;
}

```

In our FA supplement example it makes sense to assign the family classification value to the nuclear family head. We do so where (about line 368) the SPSM\GLASS version of Afamod.cpp assigns other values to the eldest member. That original Afamod.cpp code appears as --

```

/**
 * Associate the taxable amount of family allowances, and the number of
 * family allowance children, with the eldest in the nuclear family.
 * The function txinet will reassign to the spouse if necessary.
 */

nf->nfineld->im.imtfa = tfa;
nf->nfineld->im.imqtfa = qtfa;
nf->nfineld->im.imnfach = (NUMBER) nch;

```

After our addition, the modified code reads --

```

/**
 * Associate the taxable amount of family allowances, and the number of
 * family allowance children, with the eldest in the nuclear family.
 * The function txinet will reassign to the spouse if necessary.
 */
nf->nfineld->im.imtfa = tfa;
nf->nfineld->im.imqtfa = qtfa;
nf->nfineld->im.imnfach = (NUMBER) nch;
/* assign family classification by number of supplement children to the
nuclear family head */
nf->nfin->im.uv.uvfclfasup = ncfasup;

```

Finally, of course, we need to make sure that the variables for the supplement and the number of children supplemented are assigned to the mother if feasible (or in the absence of the mother to the head of the nuclear family). The relevant original SPSM\GLASS Afamod.cpp code appears as follows --

```

/* assign FA to mother if present */
if (nf->nfspoflg && (nf->nfinspo->id.idsex == FEMALE)) {
    DEBUG1("%s spouse is the mother\n");
    in = nf->nfinspo;
}

else {
    DEBUG1("%s head receives FA\n");
    in = nf->nfineld;
}

```



Our changes to this are minimal. We add only two new lines to assign the amount of the supplement and the number of supplemented children. Note that we are assigning the values of the intermediate variables to the (fully qualified) user-defined variables that we defined via `vsu.h` and `vsdu.cpp` above. The modified version of the source code reads as follows:

```
/* assign FA and the supplement, and # Fa supplement children to the mother when
she is present */
    if (nf->nfspoflg && (nf->nfinspo->id.idsex == FEMALE)) {
        DEBUG1("%s spouse is the mother\n");
        in = nf->nfinspo;
    }
    else {
        DEBUG1("%s head receives FA\n");
        in = nf->nfineld;
    }

    in->im.iffa = ffa;
    in->im.impfa = pfa;
    in->im.imqaafa = qaafa; /* Quebec Availability Supplement */
    in->im.imqnbfa = qnbfa; /* Quebec Newborn Allowance */
    in->im.uv.uvfasup = fasup; /* assign new supplement */
    in->im.uv.uvncfasup = ncfasup; /* assign # of children */
```

## Compilation

We should debug the model and test if it work properly and then compile the new model `GLASSEX3.EXE`.

## VALIDATION

Once the compilation is complete and the `GLASSEX3.EXE` file exists, the user can validate it to check whether the logic is performing as was intended. Since validation was illustrated in some length in Section 6, we include here only one illustrative set of crosstabulation outputs. In everyday operation, the user will want to ensure the correctness of the model before proceeding on to make production runs of the desired tables.

The mini-validation here consists of one set of tables for a single parameter configuration. It uses the 1986 version of the SPSM, and models the tax and transfer system existing in 1986. The user sets up the control parameter file to use `C:\SPSD\BA86.MPR` as the base system model file. The variant system, the one using the new logic for Family Allowances, is here named `GLASSX3A.MPR`. It calls for a subsidy of \$120 per year for the second and subsequent children aged 0 to 17 in the nuclear family. The relevant XTSPEC appears as follows:

```
XTSPEC

NF: uvfc1fasup+ *
    {units,
      imffa: L="New Family Allowance",
      _imffa: L="Base Family Allowance",
      uvfasup: L="New FA Supplement"};
NF: nfnkids+ *
    {units,
      imffa: L="New Family Allowance",
      _imffa: L="Base Family Allowance",
```

```

imffa-_imffa: L="Family Allowance Increase"};
NF: nftype+ *
{uvfasup: L="New FA Supplement",
immdisp-_immdisp: L="Disposable Income Increase"}

```

The first table specification illustrates the use of user-defined variables as analysis and classification variables. Note that the usage is just the same as if the variables had been part of the original SPSM, even to the ability to use the "+" qualifier to indicate the aggregation across a categorical variable's dimension.

The similarity between the first two tables is intentional; it shows that one can use the created variables to display information that is less conveniently available from SPSM variables. First, for example, the user does not have to take a difference between two variables to see the pre-tax impact of the FA supplement. Second, using the uvfacfasup variable rather than the nfnkids variable allows the user to collapse across all those nuclear family units that have no children. The third table then confirms that the supplement is being taken into account by the rest of the tax/transfer system, so that, in aggregate, the families' gains in income are less than the gross amounts of supplement awarded. The tables that result, edited very slightly as to

```

SPSD/M (Database 4.00)
Wed Sep 27 08:34:51 1989
Base Description: 1986 actual
[Driver: Version 4.00: 82-89, File: c:\spsd\ba86.mpr]
Variant Description: 1986 actual
[Driver: FA Suppl New Vars Ex, File: glassx3a.mpr]
Sample: 0.0495
AGENAME='Standard adjustment'

```

Table 1U: Selected Quantities for Nuclear Families by Family Class for FA Supplement

Family Class for FA Supplement	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	New FA Supplement (M)
0 Ch	10621.5	564.1	564.1	0.0
1 Ch	1196.5	1020.3	876.7	143.6
2 Ch	521.8	758.2	633.0	125.2
3 Ch	81.1	160.3	131.1	29.2
4 Ch	14.6	34.6	27.6	7.0
5 Ch	1.5	4.3	3.4	0.9
6 Ch	0.0	0.0	0.0	0.0
7 Ch	0.0	0.0	0.0	0.0
All	12437.1	2541.7	2235.8	305.9

Table 2U: Selected Quantities for Nuclear Families by Number of children in nuclear family

Number of children in nuclear family	Unit Count (000)	New Family Allowance (M)	Base Family Allowance (M)	Family Allowance Increase (M)
--------------------------------------	------------------	--------------------------	---------------------------	-------------------------------

0	9042.2	0.0	0.0	0.0
1	1579.4	564.1	564.1	0.0
2	1196.5	1020.3	876.7	143.6
3	521.8	758.2	633.0	125.2
4	81.1	160.3	131.1	29.2
5	14.6	34.6	27.6	7.0
6	1.5	4.3	3.4	0.9
7	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0
+-----+				
+-----+				

Table 3U: Selected Quantities for Nuclear Families by Nuclear family type

Nuclear family type	New FA Supplement (M)	Disposable Income Increase (M)
+-----+		
With Kids, 1 Adult	22.2	17.8
With Kids, 2+ Adult	283.7	179.3
With Elderly, 1 Adult	0.0	0.0
With Elderly, 2+ Adult	0.0	0.0
Other, 1 Adult	0.0	0.0
Other, 2+ Adult	0.0	0.0
+-----+		
All	305.9	197.0
+-----+		

Note that the first and second tables are completely consistent, except that the first is slightly more compact (with fewer lines), marginally better labeled, and a bit easier to specify in XTSPEC. In terms of substance, however, the two are comparable; the rows from "1 Ch" to "7 Ch" in the first table contain exactly the same information as the "2" to "8" rows of the second table. This sameness of content is just what we would expect for an option that subsidizes the second and subsequent children. The first and second rows of the second table, tabulating families not eligible for any supplement, collapse into a single line in the first table.

The third table shows that some of the Family Allowance is being recovered, since the increment in disposable income is less than the full amount of the new supplement. Further, the fraction "recovered" via the reactions of other programs in the tax/transfer system is, as expected, greater for two-parent units than for single-parent units.

Once the validation is complete, the user will proceed to the production of the desired tables and other outputs.

## SUMMARY/CONCLUSIONS

We summarize this chapter's key points by providing a checklist of the main items required to add new user-defined variables to an SPSM model.

1. Plan the desired changes "on paper". Choose the new variable names and lay out the logic by which they will be derived. Ascertain which specific substantive source code

files will be affected (e.g. `Afamod.cpp`). Choose a subdirectory for the new model, creating it if necessary.

2. Copy over the relevant files to the subdirectory where the work will be done.
  - The files `SPSMGL.dsw`, `Adrv.cpp`, `vsu.h`, and `vsdu.cpp` will always be needed, along with the relevant substantive files, e.g. the `Afamod.cpp` file of our example.
  - The files `mpu.h` and `ampd.cpp` may also be needed depending on whether parameters are to be added at the same time.
3. Update project and change the name of the output file.
4. Update `Adrv.cpp`.
  - Insert appropriate short descriptions for the two documentary string arguments (`ALTNAME` and `Tdrv`).
  - Change the function calls to refer to the alternate versions of the tax/transfer calculation functions, e.g. `Afamod(hh)` rather than `famod(hh)`.
5. Update `vsu.h`. Inside the `'uv_'` structure, indicate the types and names of the new user-defined variables. Remember to use the `'uv'` prefix, but to omit any leading underscore.
6. Update `vsdu.cpp`.
  - For each new variable, provide a `vardef` function call to define the nature of the variable to the SPSM.
  - Also for each new user-defined variable, invoke `stradd` to provide a variable description (text string) for the variable.
  - For each integer variable, analysis or classificatory, invoke `stradd` a second time (using just the stem name) to provide a list of labels for the integer values of the variable. Remember that for the analysis variables these only indicate the number of categories (from 0 to n), while for integer classification user-defined variables, the labels are text of the user's choosing.
7. Make the necessary changes to the substantive tax/transfer routines. Consider using intermediate variables to simplify things. Be careful to perform appropriate initializations and to assign the derived values to an appropriate individual.
8. Compile the new model. Don't forget to validate it before using it for any serious production work.

## Changing Base and Variant Data Variables

This chapter describes how users can, when appropriate, change values in the SPSD/M database for the analysis of policy options. Such changes stand in contrast to the changes in model logic, parameters and dependent variables described in previous chapters. Here, we are looking at changes to the data used as input by the tax/transfer algorithms rather than to the logic of those algorithms. The kinds of changes discussed here are temporary. They

affect the values "seen" by the user's model in a particular run, but they do not affect the values actually stored in the SPSD itself.

Typically, but not exclusively, the user's database changes will involve dollar-denominated amounts -- income or deduction items. The user might wish to grow or shrink income from a particular source, e.g. shrinking interest income to reflect an assumption about falling interest rates. However, the user might also want to alter a non-income variable, e.g. the school attendance variable for older children in selected families.

For SPSM models that simulate two (base and variant) tax/transfer systems, an important distinction is whether the changes affect the values as "seen" by the user's entire model, or by just one of the (base or variant) systems within the model. This distinction is so important that we have organized the structure of this chapter around it. Note, however, that the distinction is irrelevant for models that simulate only a single tax/transfer system. The procedures recommended here encourage the user to apply the single system approach whenever it is feasible.

The following section describes how to make alterations to the data right after the SPSM has read it for a model run. The changes discussed there will naturally affect ALL of the tax/transfer systems appearing in the model. The section describes two sub-cases -- In the first subcase, the user makes the data adjustments via the SPSM's built-in data-aging facilities. In the second, more demanding, subcase, the user crafts his/her own aging logic. This second subcase may involve the definition of new data-aging parameters for the model. The first section indicates where and how to make "single system" changes, and provides a detailed worked example.

The subsequent section, in contrast, describes changes that affect only a single system (base or variant) within an SPSM run. It explains how the use of the SPSM's "results file" facility can often turn this case into the simpler "single system" as described in earlier on. However, for instances in which the results file approach is impossible or inconvenient, this section also includes a description of where and how to make the necessary changes. It concludes with a worked example of how to implement system-specific database adjustments.

## **MAKING CHANGES THAT AFFECT ALL TAX/TRANSFER SYSTEMS IN A MODEL:**

This section describes how to make data changes that affect all of the tax/transfer systems in an SPSM model. It is appropriate both when the model has only a single tax/transfer system and when the model has two systems, but the user wants the data changes to affect both of them.

This section first examines the SPSM's built-in data-aging facilities. Under this method, the user assigns values to existing aging parameters via API (Aging Parameter Include) files.

This is followed by adding new data adjustment algorithms. For this type of aging, the user will define the new aging logic in the `adju.cpp` file, and will probably define new parameters via changes in the `apu.h` and `apdu.cpp` files. The user may also wish to define new dependent variables to assist in model validation.

Lastly a detailed worked example for this second subcase is presented followed by a checklist for making this “global” data aging type of change.

### **Typical Income and Population Growth Changes Via APR/API Files**

The design of the SPSD/M already anticipates the user's typical data-aging needs. The \SPSD subdirectory includes a number of files with names of the form BAxx\_yy.APR that instruct the SPSM to age the data, other than the underlying demographic structure, from year XX to year YY. Thus, file BA86\_88.APR contains the aging parameters to age the SPSD's non-demographic variables from 1986 to 1988. The degree of detail for this aging is considerable. Each of these files contains some 600 plus numeric parameters that are used by the SPSM's built-in aging algorithms.

If the substance of the parameters in these files is acceptable to the user's needs, then the data aging is straightforward. The user enters the name of the "most nearly correct" file as the control parameter file's INAPR parameter. Any necessary changes to these parameter values are then implemented via an ".API" (Aging Parameter Include) file.

The *Parameter Guide* provides the authoritative description of these parameters. However, it is useful here to characterize broadly the extensive control they provide.

Some parameters specify how imputed/converted incomes are to be treated (i.e. ignored or either of two synthesis methods adopted). A large block of parameters governs the "removal" of commodity taxes from family expenditures.

Another parameter block provides the low-income cutoffs for families. It permits the user to specify a set of "poverty thresholds" for economic families, with the particular thresholds varying by family size and the size of place of residence. Probably of most value to a typical user, though, is the large set of growth factors for the SPSD's dollar-denominated data variables: incomes, deductions, and expenditures. Virtually every such variable has its own growth factor.

The SPSD/M also provides for convenient demographic aging of its underlying population. The SPSD directory's ".WGT" files provide the user with the capacity to adjust the population base throughout the interval 1984 to 1991.

### **Changes Involving New Logic For `adju.cpp`**

The flexibility provided by the aging parameter (".APR" & ".API") and population aging (".WGT") files will often be sufficient for the user's needs. However, in some circumstances, the user will wish or need to exercise more direct control over the data to be used for a simulation. A few examples will indicate the scope of what is possible. The reader should appreciate that the focus of these examples lies more in quickly conveying that scope than in maintaining a strict, policy-oriented realism.

1. The user could increase the average education level by adjusting the "idedlev" variable for selected individuals, perhaps resulting in a distribution of educational attainments that falls in line with some exogenous forecast.

2. The user might wish to grow some income or transfer amount by a factor that is a function of the unit's characteristics. E.g., based on the assumption that investors' portfolios differ as a function of investor age and income, a user might be unwilling to model the effect of an increase in interest rates by growing everyone's interest income using the same proportion. Instead, a smaller factor might be applied to those individuals felt likely to be conservative and/or to have portfolios that turn over more slowly. This type of assumption would treat such families as being unable to benefit as quickly from the higher interest rates.
3. A user might wish to model greater labour force participation by changing the array of labour force variables relevant for individuals in the SPSD (weeks worked, paid-employment earnings, unemployment insurance variables, etc.). Changes in such a wide variety of related variables would only be done after considerable, comprehensive planning.
4. At the extreme, a highly experienced, knowledgeable SPSM user could even alter the household/family structure of the SPSD, modeling a baby boom by adding "synthetic children" to appropriate families in the database.

The `adj_u.cpp` function, found in the `\SPSM\GLASS` subdirectory, is the means by which the user can add new data aging logic to SPSM models. That `adj_u.cpp` function is called immediately after the SPSM has read in each household, and before any transfers or memo variables have been computed. The user can insert the logic for his/her own changes immediately after the "adj(hh)" invocation that the SPSM uses to carry out its own data aging, i.e. its built-in application of the income growth parameters specified in the relevant ".APR" and ".API" files.

For the implementation of new data aging logic, users may need to define new intermediate variables (including counters, pointer variables, etc.) and/or to define new, custom, data aging parameters. The next subsection describes the general procedure for adding such new data aging parameters, with the attendant specific changes developed in the worked example that follows it.

### **Adding New Database Adjustment Parameters**

The addition of new user-defined database parameters closely parallels that of new model parameters as described in previous chapters. However, some minor differences are relevant.

(1) SPSM models have only a single aging parameter file (extension ".APR"); they may have either one or two model parameter (extension ".MPR") files, depending on whether they model one or two transfer systems. (2) Correspondingly, users supply the values of user-defined aging parameters in ".API" (Aging Parameter Include) files that amend standard ".APR" files, rather than via ".MPI" (Model Parameter Include) files that amends standard ".MPR" files. (3) New aging parameters are defined in the `apu.h` (header) file rather than the `mpu.h` header file used for model parameters. (4) Similarly, the function calls that make the parameters available to the rest of the model occur in `apdu.cpp`, rather than the `ampd.cpp` file used for model parameters. However, the structures of the relevant pmadent

and stradd calls are exactly identical. Note, though, that certain arguments to these functions differ between aging and model parameters. The worked example highlights these differences. (5) Finally, the logic changes proper are defined in `adju.cpp`, rather than (typically) the individual tax/transfer functions, such as `Afamod.cpp`, that are relevant for changes to the transfer calculation logic of a model.

We note in passing that the SPSM's control parameters follow a similar parallel structure, but, even in glass box applications, users do not need to `DEFINE` new control parameters. Instead, they simply alter the values of existing control parameters.

### **A Worked Example**

Our hypothetical user, seeking to reflect a response to some change in federal income tax treatment, wants to grow RRSP contributions in a model. S/he wants the growth to apply either for a single system to be analyzed, or for both the base and variant systems in a comparative model. However, this user is not willing to assume that everyone's contributions grow by the same rate, and wants to simulate disproportionate growth as a function of income. The main focus of the model is assumed to lie elsewhere in the tax/transfer system. That is, the user has no special interest in the impacts of the RRSP increases themselves. Rather, the user just wants "better" representations of the deduction amounts to be used in all of the calculations for the relevant transfer system(s).

To make the example more precise, assume that the user wishes to grow existing contributions by  $x\%$  for each (whole or partial) slice of \$10,000 of paid employment earnings and self employment earnings over an initial base amount of \$20,000. Thus, an individual with \$45,000 in earnings would see his/her RRSP contribution grown by a factor of  $(1.0 + 3x)$ , where  $x$  is a new user-defined parameter. This growth will be **IN ADDITION TO, AND COMPOUNDED WITH**, any growth induced via the standard SPSM growth parameter for RRSP contributions, `GFRRSP`.

In a possibility **NOT** developed here, the user might also have induced the presence of RRSP contributions for individuals who reported zero such contributions. The example developed later in Section 9.2 provides an illustration of this sort of synthesis of dollar-denominated amounts.

In the remainder of this subsection, we track the individual steps involved in implementing this conditional growth (beyond the growth implemented via the aging parameters `GFRRSP`). We assume that the user has created the subdirectory `GLASSEX4` for the purpose, and "COPIED IN" all of the relevant files. (`SPSMGL.dsw`, `apu.h`, `apdu.cpp`, and `adju.cpp`, plus the SPSM parameter files relevant for running the new model). In this directory the user will create an `".API` file to provide a value for the new user-defined parameter.

Because the parameter addition process for aging parameters so closely parallels the procedure described in previous chapters, and for model parameters, our commentary on these changes is kept to a minimum. The user is assumed to have modified the project to include all the relevant files and changed the output name of the compilation to `GLASSEX4.EXE`



We include the aging documentation in the relevant string defined in `adju.cpp` as described below.

#### (A) Changes to `apu.h`

We begin by defining a user-defined parameter for the user-defined RRSP contributions growth factor, the "x" factor in the description above. As a mnemonic, `UDGFRRSP` (User-Defined Growth Factor, RRSP contributions) seems appropriate. The SPSM provides for up to 100 bytes worth of user-defined aging parameters, with this allocation independent of the 600 bytes allocated for any model parameters that the user may wish to define.

The additions to `apu.h` indicate the kind of parameter being defined. They go just before the function prototype specifications, replacing the dummy user aging parameter `UADUMMY` in the `apu.h` code.

```
typedef struct UA_ {  
int UADUMMY;      /* dummy entry */  
}  
UA_;
```

In our example, we replace the single `UADUMMY` line with --

```
NUMBER UDGFRRSP;      /* User-defined growth factor for RRSP Contr.  */
```

#### (B) Changes to `apdu.cpp`

In the `apdu.cpp` function we add invocations to the `pmaddent` and `stradd` functions to give the broader SPSM access to the value of the new parameter. The details of these functions appear in previous chapters. We make the additions at the end of the `apdu.cpp` function, right before the statement --

```
DEBUG_OFF("apdu");
```

Our two invocations look as follows:

```
pmaddent(pap, "UDGFRRSP", (char *)&AP.UA.UDGFRRSP, NULL, P_SCL, C_NUM, 0, 0, NULL, 0);
```

and

```
stradd("UDGFRRSP", "User-defined growth factor for RRSP Contr.");
```

The explanatory text at the beginning of `apdu.cpp` function describes the AGING-ORIENTED arguments for `pmaddent` and `stradd`. It also provides the templates for our utilization here (a scalar parameter).

There are two critical differences in the `pmaddent` utilization as compared with the definition of new model parameters. (1) The first argument is `pap` rather than `pcp`. (2) The third argument differs in that the new parameter resides in the `UA` (User Aging) structure within the SPSM's `AP` (Aging Parameter) structure. This stands in contrast to the "&MP.UM" reference used for user-defined model parameters (User Model within Model Parameters).

#### (C) Changes to `adju.cpp`

The first change updates the documentary text string relating to data aging. The original SPSM\GLASS function defines this string at about line 43 as

```
/*global*/ char AGENAME[IDSIZE+1] = "Unnamed";
```

We modify it here to read --

```
/*global*/ char AGENAME[IDSIZE+1] = "RRSP Contr(Earnings)";
```

With the parameter value available throughout the SPSM, we make the source code additions to implement the RRSP contribution growth. The first thing we need is some local variables to aid us in stepping through the individuals in the household being analyzed and possibly assigning modified RRSP contributions. Thus, we add the following four declarations to the `adju.cpp` function, inserting them just after the function's opening brace.

```
NUMBER earn;          /* total paid and self-employment earnings */
int group;             /* number of UDGFRSP multiples to use */
register P_in in;      /* pointer to data for current person */
int ini;               /* persons processed */
```

For the aging assignments themselves, the relevant location is near the very end of the `adju.cpp` function, inside the code segment --

```
    DEBUG_ON("adju");
    /* Just call the standard adjustment algorithm */
    adj(hh);
    DEBUG_OFF("adju");
```

Our addition goes between the `adj(hh);` and `DEBUG_OFF("adju");` statements.

```
/* Grow RRSP contributions as a function of total earnings */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    if (in->id.idrrsp == (NUMBER)0.0) {
        continue;
    }
    earn = in->id.idiemp + in->id.idisefm + in->id.idisenf;
    if (earn <= (NUMBER)20000.0) {
        continue;
    }
    group = (int)(ONE+(earn-(NUMBER)20000.0)/(NUMBER)10000.0);
    in->id.idrrsp*=(ONE+AP.UA.UDGFRSP*(float)group);
}
```

The new code, headed by an explanatory comment, breaks out into components that are relatively straightforward.

- (1) The control portion of the "for" statement has been copied, in its entirety, from the `memo1.cpp` function (computing totals for individuals) in the SPSM/GLASS subdirectory. It steps across the individuals in the household. The local variables defined earlier are used in this stepping.
- (2) Growing RRSP contributions multiplicatively is not meaningful if there are none to begin with. Thus, the "if-continue" statement of the next three lines skips the remainder of the four statements if the individual has no RRSP contributions. The typecast "NUMBER," here and later, indicates the user's intentions as regards variable types; it prevents compiler warnings.
- (3) If RRSP contributions are positive, the next line calculates the individual's earnings from paid employment and from farm or non-farm self-employment. If the total does not exceed \$20,000, then the remainder of the for statement is skipped; Another "if-continue"

statement performs this function.

- (4) The assignment to the "group" variable computes the number of multiples of UDGFRRSP relevant for the growth. The final statement in the body of the loop applies the growth via a multiplicative assignment. These two statements will only be executed if some growth is appropriate. The (int) and (NUMBER) casts they contain indicate the user's explicit intentions as to variable type conversions; they serve to prevent meaningless warnings during the compilation stage.

#### (D) Compiling the model enhancement

The model should be debugged before the compilation of GLASSEX4.EXE executable file. Only then can the model be run for validation testing and production work.

#### (E) Providing a parameter value

For any particular run of the model, the user must provide a value for the new parameter, e.g. a value of 0.01. Normally the user will do this "on the fly" during the run of the new model, or via an ".API" (Aging Parameter Include) file that will modify the contents of the APR file specified in the model's control file (".CPR"). In our example, the ".API" file would consist of the single line --

```
UDGFRRSP      0.01
```

if no existing aging parameters were to be modified.

#### (F) Validating the model

Before using the model at all seriously, the user would want to validate the model to be sure it is performing as intended. Though we shall not carry out such a validation in detail here for reasons of space, normally one would generate a few selected tables for different runs, checking to ensure that the model produces the expected results. For example, inputting a UDGFRRSP factor of zero should leave the total amount of the RRSP unchanged. Similarly, a small value, say 0.01, should have a small or zero effect on low-income units, but should have a larger effect on higher income units. A table, defined at the level of the individual, that showed the increase in the RRSP contributions variable as a function of individual earnings would go a long way toward deciding whether the algorithm yields the right amount of RRSP increase. It could be generated by using a results file based on the unmodified database, and comparing numbers of individuals and amounts of RRSP contributions to the counterparts of these variables after the new RRSP contribution aging.

When one uses the 5% sample SPSPD, together with the 1986 population, aging parameters, and model parameters, one gets the following summary results for a UDGFRRSP factor of 0.01:

	Before Growth	After Growth	Difference
RRSP Contributions (M\$)	11,134.3	11,329.2	194.9
Federal Income Tax (M\$)	41,173.3	41,118.0	55.3
Prov. Income Tax (M\$)	24,190.6	24,160.5	30.1

Total RRSP contributions have risen by about 1.75%, and federal and provincial income taxes have correspondingly dropped by somewhat less than the amount of new RRSP contributions.

### **Checklist for Changing Database Variables "Globally"**

(A) Check to see whether the SPSM's existing facilities are sufficient to implement the desired data aging, so that no new logic is required.

Can the desired population aging be implemented via a selection among existing case weight files? If so, then specify the relevant case weight file (".WGT" extension) via the INPWGT (Input Weight) control parameter. Use a ".CPI" file to provide the desired INPWGT value, or enter it on the fly in response to the model's prompts.

Can the adjustment of the data values be accomplished via changes to the values of the SPSM's data aging parameters, in conjunction with the SPSM's normal data aging algorithm (adj(hh))? If so, then provide the relevant aging parameter values to the SPSM via an ".API" file. Specify it to the SPSM either interactively or via a batch file being used to coordinate execution of the model.

(B) If the desired adjustments to the data cannot be handled via the built-in data aging procedures, then some new logic will be required. The steps for adding this new data aging logic are as follows:

1. Copy all of the relevant files to a new directory established for the analysis. The files \SPSM\GLASS\adju.cpp, SPSMGL.dsw are always relevant. The files \SPSM\GLASS\apu.h and \SPSM\GLASS\apdu.cpp will be relevant when new aging parameters are required.
2. Alter the project environment to include all the relevant files and change the name of the compiled model. Alter apu.h if new data aging parameters are being defined.
3. Alter apdu.cpp if new data aging parameters are being defined. The changes will consist of adding new pmaddent and stradd invocations so that the substance of the new parameters is available throughout the SPSM. Debug the model.
4. Alter adju.cpp. First change the function's documentary text string, AGENAME[IDSIZE+1]. Then implement the new data aging logic. This step will often involve declaring useful local variables and stepping through individuals or families in the household.
5. Compile and validate the model before using it for production runs. Parallel tabulations of relevant individuals and amounts before and after the data aging alterations are recommended.
6. Carry out production runs using the new, validated aging logic.

### **MAKING CHANGES THAT AFFECT ONLY THE BASE OR ONLY THE VARIANT**

Building a model in which data aging differs between a base system and a variant system is inherently more complicated than building one in which the two systems are treated identically. When it is possible, the user should avoid such complication. The SPSM's capacity for using "results files" (extension ".MRS") provides the major mechanism for avoiding data aging that is system-conditioned.

The basic approach is to divide the problem into two parts, one for each system. Then, within each such system, a single data aging algorithm applies, and the methods described earlier in this chapter. The user first creates a results file for one of the two systems, choosing the variables necessary for any system specific tabulations and for any comparisons to be made. In creating this first system, the user applies the data aging assumptions relevant to that system. Subsequently, the second system is simulated, with the appropriate, alternative, data aging applied to it. The results file is read-in, in parallel with the processing of the second system, so that the two systems, with their different data aging assumptions, are available simultaneously for all required comparisons. *Introduction and Overview Guide* provides an illustration of the use of results files.

The remainder of this section is relevant when the results file approach is somehow judged inappropriate or inadequate to the task at hand. A few examples will illustrate such circumstances.

1. The user may place a high premium on having a model that is self-contained, and, once it has been validated, relatively easy to use interactively.
2. The intended application of the model may involve sensitivity analysis that would require several MRS files, with an inherent possibility that confusion might arise. It might require, for example, investigation of the impact of altering the aging of one specific variable, with a variety of other variables repeatedly changed in parallel between the base and variant systems.
3. The intended application might involve complicated comparisons requiring large .MRS files (or many of them simultaneously) when disk storage is at a premium.

We believe, however, that these kinds of situations, while occurring occasionally, will be the exception rather than the rule. We encourage users to seek to avoid parallel system models in which data aging differs across the two systems.

In broadest outline, the method for making system-specific data changes is similar to that used to make changes to the TAX/TRANSFER LOGIC of a system. Any new system-specific data aging parameters are added, via the `mpu.h` and `mpdu.cpp` files, as MODEL parameters, and NOT as data adjustment parameters per se. As described below, the user may wish to add new MODEL dependent variables to track the changes being made. Although, if new parameters and dependent variables are not required, the procedure applies equally to SPSM base and variant models, we shall explain the procedure in terms of the more common situation of variant models.

The MODEL-oriented approach just summarized is mandated by the design of the SPSM.

Since there is only a single ".APR" file, its parameters inevitably affect the data aging for all systems within a model. In contrast, changes made via ".MPI" files, and via the system-specific `Adrv.cpp` and `drv.cpp` functions, apply only to a single designated tax/transfer system. The user can take advantage of this system-specificity to implement system-specific data adjustments.

The key to the system-specific data aging changes lies with alterations made to the `Acall.cpp` file. In essence, the user "intercepts" a household's data record just before it is used by the functions in that procedure, makes the desired changes, and later restores the data record to its original state just before execution leaves that procedure. The next section explores these `Acall.cpp`-oriented steps in greater depth.

### **Implementing Changes in `Acall.cpp`**

The focus in this section rests almost exclusively with the details of changes made within `Acall.cpp`. Because of the similarity of system-specific data adjustments to the kinds of tax/transfer system revisions described earlier in this *Programmer's Guide*, certain topics are not repeated here. Specifically, users are expected to add any new parameters, and any necessary new dependent variables using the methods documented in previous sections. For example, a user might wish to add a new model variable to indicate whether the original database value for a variable has been changed by the system-specific adjustments.

We'll take up the required changes in the order in which a reader would encounter them when reading `Acall.cpp`'s source code. Later, a worked example provides a concrete application of the changes.

#### **(A) Declare New Local (to `Acall.cpp`) Variables**

Recall that the general procedure requires the user to save the values of the variables to be adjusted. The storage permits the values to be restored again before leaving `Acall.cpp`. Thus, the user must include in `Acall.cpp` appropriate local declarations to provide the needed storage. Typically, the variables to be adjusted will be defined at the level of the individual. Thus, the new variables should typically be defined as vectors of length `MAXIND`. (`MAXIND` is the maximum number of individuals in a family.) The user may also wish to define other local, working, variables. Normally, the user will declare these variables just before the opening brace for the function, at about line 99 of the unmodified version of `Acall.cpp`.

#### **(B) Save the Values to Be Changed**

As the very first thing within the executable portion of `Acall.cpp`, the user should store away the original values of the variables that will be changed. If this is done, none of the other functions invoked inside `Acall.cpp` can alter the value first or use the unaltered value. Typically, the storage is accomplished via a "for" statement that steps across the individuals in a household and copies them, one at a time, into the elements of a vector declared in step (A). One of the elements in the bestiary provides the relevant stepping control. The user will do this at about line 90 of the unmodified code, just after the statement -

```
DEBUG_ON( "Acall" );
```

### (C) Change the Database Values

Immediately after the values have been stored, and still before the household's pointer has been passed to any of the tax/transfer or roll-up functions, the user should make the desired changes to the values of the relevant variables. These changes will constitute the bulk of the "real programming", i.e. logic that cannot necessarily be conveniently adapted from elsewhere in the SPSM.

### (D) Use the Now Adjusted Values

This step is the easiest of all, since it requires no special effort on the part of the user. It consists of **RETAINING** the calls to the several tax/transfer and memo functions. Since the values of the relevant variables have already been adjusted at this point, all of those functions will perform their calculations using the adjusted household.

### (E) Replace the Original Values

The final step consists of restoring the original values to the variables that were adjusted. It will typically be done at about line 99 of the unmodified version of `Acall.cpp`, just before control passes out of the function, i.e. just before the statement --

```
DEBUG_OFF( "Acall" );
```

Execution of the replacement is important from the perspective of the code's generality, maintainability, and reusability. The user programs the changes without knowing whether the system programmed will be a base or variant system. By putting things back the way they were, the user can minimize the possibility of unwanted side effects elsewhere in the model. Equally important, this procedure minimizes the potential for unwanted side effects should the new adjustments be used again in another model.

## **A Worked Example**

### (A) The Substance to be Modeled

We begin with a description of the substantive logic used in the example. It will be obvious that the same data-aging goals could have been achieved using the "avoidance" techniques described above; however, since our documentary objective here is the illustration of system-specific data aging techniques, we arbitrarily deem those avoidance techniques to be "inappropriate" for our immediate purposes.

Suppose that some exogenous analysis relating to new income tax reporting requirements suggests that individuals will be reporting more self-employment income. More specifically, suppose that 5% of those individuals (1) not reporting more than \$100.00 of self-employment income (farm and non-farm combined) and (2) who are aged both over 25 and under 60 and (3) who further have half a year or more without work and looking for work, really have non-farm self-employment income that has not previously been reported, but now will be reported. Moreover, suppose the amounts of "new" self-employment income for these persons is believe to be distributed uniformly between zero and \$4000 per year.

The user seeks to estimate the additional income taxes collectible from these persons and also to assess the impact of this "discovered" income on reducing the poverty rate as measured against the LICOs. To carry out this investigation the user plans, in the variant tax/transfer system, to impute appropriate amounts of these new incomes to randomly selected persons who satisfy the three conditions.

#### (B) Relevant New Parameters and Variables

Following recommended SPSM practices for avoiding hard-wired values in a model, the user establishes the following new user-defined aging parameters:

Parameter	Description:	Value:
NSEFLAG	"New Self-Employment Income Flag"	1
NSEAMT	"New Self-Employment 'Trivial Amount'"	100.0
NSEFRC	"New Self-Employment Fraction"	0.05
NSEWKS	"New Self-Employment Weeks Requirement"	26
NSEMINAGE	"New Self-Employment Minimum Age"	25
NSEMAXAGE	"New Self-Employment Maximum Age"	60
NSEMAXINC	"New Self-Employment Maximum New Income"	4000.0

Similarly, the user defines new variables that will permit convenient counts of the numbers of eligible persons and of the number for whom new incomes are synthesized. It will also be useful to have an additional new variable for the amounts of synthesized income.

Variable: Description:

uvnseef	"Eligible New Self-Empl"
uvnseesf	"Received New Self-Empl"
uvnseamt	"New Self-Empl Amount"

#### (C) Setting Up for the Analysis

The user begins by creating a new subdirectory for the analysis, GLASSEX5. S/he copies in the required template files: SPSMGL.dsw (to control the compilation), mpu.h and Ampd.cpp (to make the new parameters available), vsu.h and vsdu.cpp (to make the new variables available), and Acall.cpp (to implement the new system-specific database adjustments).

We look at the changes in the order in which the user would be encouraged to make them.

#### (D) Changes to project

All the relevant files should be included in the project and the name of the output model changed to GLASSEX5.EXE.

#### (E) Changes to mpu.h

The user provides declarations for all of the new parameters described above.

```
int    NSEFLAG;      /* New Self-Employment Income Flag */
NUMBER NSEAMT;      /* New Self-Employment 'Trivial Amount' */
```



```

NUMBER NSEFRC;      /* New Self-Employment Fraction */
NUMBER NSEWKS;      /* New Self-Employment Weeks Requirement */
NUMBER NSEMINAGE;   /* New Self-Employment Minimum Age */
NUMBER NSEMAXAGE;   /* New Self-Employment Maximum Age */
NUMBER NSEMAXINC;   /* New Self-Employment Maximum New Income */

```

#### (F) Changes to Ampd.cpp

The user alters the Ampd.cpp file by providing pmaddent and stradd invocations for all of the new parameters. Appropriate new pmaddent calls would be as follows:

```

pmaddent(pcp, "NSEFLAG", (char *)&MP.UM.NSEFLAG, NULL, P_SCL, C_INT, E_FLAG, 0, NULL,
0);
pmaddent(pcp, "NSEAMT", (char *)&MP.UM.NSEAMT, NULL, P_SCL, C_NUM, 0, 0, NULL,
0);
pmaddent(pcp, "NSEFRC", (char *)&MP.UM.NSEFRC, NULL, P_SCL, C_NUM, E_FRCT, 0, NULL,
0);
pmaddent(pcp, "NSEWKS", (char *)&MP.UM.NSEWKS, NULL, P_SCL, C_NUM, 0, 0, NULL,
0);
pmaddent(pcp, "NSEMINAGE", (char *)&MP.UM.NSEMINAGE, NULL, P_SCL, C_NUM, 0, 0, NULL,
0);
pmaddent(pcp, "NSEMAXAGE", (char *)&MP.UM.NSEMAXAGE, NULL, P_SCL, C_NUM, 0, 0, NULL,
0);
pmaddent(pcp, "NSEMAXINC", (char *)&MP.UM.NSEMAXINC, NULL, P_SCL, C_NUM, 0, 0, NULL,
0);

```

The associated stradd invocations would appear as follows:

```

stradd("NSEFLAG", "New Self-Employment Income Flag");
stradd("NSEAMT", "New Self-Employment 'Trivial Amount'");
stradd("NSEFRC", "New Self-Employment Fraction");
stradd("NSEWKS", "New Self-Employment Weeks Requirement");
stradd("NSEMINAGE", "New Self-Employment Minimum Age");
stradd("NSEMAXAGE", "New Self-Employment Maximum Age");
stradd("NSEMAXINC", "New Self-Employment Maximum New Income");

```

#### (G) Changes to vsu.h

In this file the user declares the new variables that will contribute to more convenient validation and tabulation of the individuals for whom new income is considered or actually synthesized.

```

int      uvnseef; /* Eligible for New Self-Empl Synthesis */
int      uvnseef; /* Received New Self-Empl Income */
NUMBER   uvnseamt; /* New Self-Empl Amount */

```

#### (H) Changes to vsdu.cpp

In vsdu.cpp the user invokes vardef and stradd to make the new variables available throughout the new model. As indicated above, there are two classificatory variables to be used for crosstabulation outputs, and a NUMBER float value for the amount of synthesized self-employment income.

```

/* uvnseef: (Class) Flag: Individual eligible for NSE synthesis? */
vardef("_uvnseef", IN, im.uv.uvnseef, C_INT, V_CLAS);
stradd("uvnseef", "Eligibility for Synth Self-Empl");
stradd("nseef", "\tNot Eligible\tEligible");

/* uvnseef: (Class) Flag: Individual Got Synth. NSE? */
vardef("_uvnseef", IN, im.uv.uvnseef, C_INT, V_CLAS);
stradd("uvnseef", "Synth Self-Empl Receipt");
stradd("nseef", "\tNo Receipt\tReceipt");

```

```

/* uvnseamt: (Analysis) NUMBER: Amount of synthesized NSE */
vardef("_uvnseamt", IN, im.uv.uvnseamt, C_NUM, V_ANAL);
stradd("uvnseamt", "Synth Self-Empl Amount");

```

### (I) Changes to Acall.cpp

i) The changes begin with the declaration of new variables critical to the data adjustment process. We use standard SPSM notation for the pointer to an individual, and for the number of persons processed (for the stopping rule within households). In addition, there is a vector declared to hold the original values of the individuals' non-farm self-employment income.

```

register P_in in;          /* pointer to data for current person */
int ini;                  /* persons processed */
NUMBER orignfse[20];      /* original non-farm self-empl income */

```

ii) The changes continue with the code to store the existing non-farm self-employment income so that it can later be restored to its original state. We use one of the standard elements of the bestiary, stepping across individuals in the household, to implement this archival.

```

/* Archive original database values for non-farm self-employment */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    orignfse[ini]=in->id.idisenf;
}

```

A slightly more efficient version of this code would make the execution of the storage instructions conditional upon the NSEFLAG parameter being set to a value of 1 to activate the synthesis facility. The version here is simpler and slightly safer.

### iii) Implement the conditionally augmented self-employment income

[Work in the use of existing pseudo-random variables for both the choice of new persons to report self-employment earnings (non-farm) and the amount of it to report. Explain how this is central to replicability given selection of subsets of the data.]

```

/* Selectively synthesize non-farm self-employment income */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {
    in->im.uv.uvnseef=0; /* assign values to new vars */
    in->im.uv.uvnseef=0;
    in->im.uv.uvnseamt=(NUMBER)0.0;
    if (MP.UM.NSEFLAG==0) {
        continue; /* don't synthesize if facility is off */
    }

    if ( ((in->id.idisefm+in->id.idisenf)>MP.UM.NSEAMT) ||
        (in->id.idnage<MP.UM.NSEMINAGE) ||
        (in->id.idnage>MP.UM.NSEMAXAGE) ||
        (in->id.idlyun<(int)MP.UM.NSEWKS) ) {
        continue; /* ignore ineligible individuals */
    }

    in->im.uv.uvnseef=1; /* mark indiv. as potentially eligible */

    if (in->id.idrand[2]>MP.UM.NSEFRC) {
        continue; /* individual was not selected to get income */
    }
}

```

```

in->im.uv.uvnsef=1; /* mark indiv. as recipient */ in->im.uv.uvnseamt=in-
>id.idrand[3]*MP.UM.NSEMAXINC; /*synthesize amt */ in->id.idisenf+=in-
>im.uv.uvnseamt; /* add syn amt to non-farm self-empl */
}

```

The preceding code, though a bit lengthy, is straightforward. Inside the loop through individuals, one performs the following actions:

Assign default values to the new user-defined variables.

Skip the rest of the loop if the facility was not activated.

Skip the rest of the loop if the individual doesn't meet the qualifying conditions for synthesis of new self-employment income.

Mark the individual as potentially eligible for synthesis; then skip the rest of the loop if the individual is not "chosen" to receive income.

If execution reaches this stage, mark the individual as a recipient of synthesized income and impute the amount, adding the new amount to the person's non-farm self-employment variable.

Once the loop has been executed, the synthesis of new non-farm self-employment income is complete for all members of the household. At this point the "regular" statements of `Adrv.cpp` follow, calculating the tax/transfer amounts and the several memo items.

iv) Finally, after the adjusted household has been processed through all of the tax/transfer and memo functions, the new code restores the original non-farm self-employment income values.

```

/* Restore original database values for non-farm self-employment */

for (ini=0, in=&hh->in[0]; ini<hh->hhnin; in++, ini++) {

in->id.idisenf=orignfse[ini]; }

```

A slightly more efficient version of this code would make the execution of the restoration instructions conditional upon the `NSEFLAG` parameter being set to the value of 1 that activates the synthesis facility. The version here is simpler and slightly safer.

## (J) The new MPI and CPI files

It still remains to provide values to the several parameters so that the SPSM, during a particular run, can implement the desired adjustments. A parameter "include file" (extension ".MPI") with the following entries performs this function.

NSEFLAG	1
NSEAMT	100.0
NSEFRC	0.05
NSEWKS	26.0
NSEMINAGE	25.0
NSEMAXAGE	60.0
NSEMAXINC	4000.0

Similarly, it is necessary to make sure that the relevant independent streams of pseudo-random variates are generated to serve as inputs to the "random" choices of synthetic income recipients and the associated amounts of synthesized income.

### (K) Compiling and Validating the Model

With all of the source code changes complete, the user should first debug the model and then compile the desired executable file, GLASSEX5. We conclude this worked example by characterizing a very quick and dirty set of validation tables. For a serious application, the user would normally undertake a much more rigorous validation of the changes. Recall too, that this kind of system-specific data adjustment could more easily have been accomplished using results files (".MRS"). Under that mechanism, an equivalent income assignment logic would have been applied via the adju.cpp file, and the relevant parameters would have been supplied via an API file.

Assume, for purposes of this quick and dirty illustrative validation, that the user's exogenous source has already indicated roughly how many individuals should display new self-employment income, perhaps as a function of some relevant policy variable.

The user will first want to tabulate the numbers of individuals according to the values of the two user-defined classificatory variables, uvnseef, and uvnseef. Then the entries in this table can be compared to the exogenous source to confirm (1) that the numbers of eligible individuals agrees with those specified in the "exogenous source," (2) that an appropriate proportion of these individuals have had new self-employment income imputed.

Next, the user would want to confirm that the average amount of new imputed self-employment income is appropriate (i.e. half of the \$4,000 NSEMAXINC parameter value). It would also make sense to tabulate the total amount of new income imputed, so that this amount can be compared to the increases in federal and provincial income taxes. Thus the user can confirm whether an appropriate proportion of the new income is flowing to the government sector as income taxes.

Even for the validation runs, it makes sense to look at the degree of change in the incidence of units below the relevant LICOs. Given the relatively tight conditions for the eligibility to receive the synthesized income, and the relatively small portion of the eligible population selected to receive new self-employment income, the user should expect only a small change in that incidence.

Here we show the first part of this validation, verifying the amounts of new self-employment income. We use the SPSPD/M for 1986 with the 5% sample. The changes in "poverty rate," not shown here, would be derived using the SPSPD's "efpovthr" (poverty threshold) and "impovinc" (income for comparison against the relevant poverty threshold) variables. The validation is most conveniently performed via crosstabulations. The relevant control parameters, input via a ".CPI" file, are as follows:

```
XTFLAG      1
```

```

XTSPEC
IN: { units }
* uvnseef
* uvnseef;
IN: { uvnseamt,
      uvnseamt/units }
* uvnseef;
IN: { uvnseamt,
      imtxf-_imtxf,
      imtxp-_imtxp }
* uvnseef

```

The resulting tables then appear as --

Table 1U: Unit Count (000) for Individuals by Eligibility for Synth Self-Empl and Synth Self-Empl Receipt

Synth Self-Empl Receipt

Eligibility for Synth Self-Empl	No Receipt	Receipt
Not Eligible	23351.7	0.0
Eligible	809.6	47.2

Table 2U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
uvnseamt/units	0	1962

Table 3U: Selected Quantities for Individuals by Synth Self-Empl Receipt

Synth Self-Empl Receipt

Quantity	No Receipt	Receipt
Synth Self-Empl Amount (M)	0.0	92.5
imtxf-_imtxf (M)	1.0	12.9
imtxp-_imtxp (M)	0.5	9.5

As regards the substance of these tables, we'll assume that the 809.6 thousand persons in table 1U agrees reasonably well with the hypothesized "exogenous data source." Since 47.2 thousand of these persons received some new self-employment income, the 5% objective has been roughly met. Presumably the proportion would be closer to 5% were we to use the full SPSPD.

Table 2U confirms that our new algorithm assigns new self-employment income only to those eligible to receive it. The total amount of new income, and the associated average amount, confirm that the expected amounts of the new income are being synthesized (roughly \$2000 per selected individual).

Table 3U then indicates how much of the new income, a bit more than a quarter of it, is being captured by the tax system. As expected, most of the capture is directly from the recipient

individuals, though there is some from non-recipients, primarily because some recipient individuals become less valuable as personal exemptions due to their new income. Clearly, with income of less than \$100M being distributed across the whole personal sector, we do not expect any major impacts on the proportion of the population below the LICOs.

Finally, once the user is satisfied as to the correctness of the adjustment procedures, s/he would run the full SPSD through the model in one or more production runs. To meet the illustrative goals described at the start of this section, outputs would have to include the federal and provincial income tax totals, and the numbers of families above and below the LICOs, with these outputs being produced both with and without the synthesis of new non-farm self-employment income. Normally, the user would also include breakouts of these variables by relevant classificatory variables such as family type.

### **Checklist for System-Specific Database Changes**

- (A) Create a new subdirectory for the analysis. Copy into it templates for all of the files that will be needed for the analysis. Items that are likely to be required include `SPSMGL.dsw`, `mpu.h`, `Ampd.cpp`, `vsu.h`, `vsdu.cpp`, `Acall.cpp`, and a control ("`.CPR`") file. The user will also create, in this same subdirectory, other files required for the analysis for which there are no obvious templates, e.g. the "`.MPI`" file that will provide values for the system-specific data aging parameters, or a batch file to control the SPSM session.
- (B) Change the project environment to include all the relevant files and change the name of the executable output file.
- (C) Change `mpu.h` and `Ampd.cpp` to declare any new system-specific data adjustment parameters, and to make them available, via invocations of `pmaddent` and `stradd`, to the rest of the SPSM.
- (D) Change `vsu.h` and `vsdu.cpp` as required to declare any new system-specific model variables, and to make them available, via invocations of `vardef` and `stradd`, to the rest of the SPSM.
- (E) Change `Acall.cpp` to save the original values of the variables to be adjusted, to effect the adjustments, and then, after the household has been processed, to restore the original values before leaving the procedure. These steps will typically require the definition of local `VECTORS` of values dimensioned for the numbers of possible individuals in a household.
- (F) Compile the new model and correct any problems identified by the compiler.
- (G) Supply values for the new system-specific data adjustment parameters via an "`.MPI`" file or files. When the aging depends on the use of pseudo-random variables, provide a "`.CPI`" file with appropriate changes to the `SEED` parameter. The model will gain access to these control and model parameter values at model execution time either interactively or via an SPSM batch file.

(H) Validate the model carefully, and then make production runs.