

**FERMI NATIONAL ACCELERATOR
LABORATORY**

ESME CODE DEVELOPMENT

by

Ayodele T. Onibokun

Supervisor: James MacLachlan

Summer Internships in Science and Technology

September 2005

FERMI NATIONAL ACCELERATOR

LABORATORY

ACCELERATOR DIVISION

ABSTRACT

ESME CODE DEVELOPMENT

by

Ayodele T. Onibokun

Supervisor: James MacLachlan

ESME is used for calculating the evolution of distribution of particles and azimuth coordinates by iterating through a map corresponding to a single-particle equation of motions. ESME, an accelerator beam simulation program is written in Fortran 77 and designed for the Accelerator division at Fermilab to simulate those aspects of beam behavior in a synchrotron that are governed by radio frequency systems. Goals of this project include successful compilation of ESME code on Fortran 90 compiler (Sun Solaris), enhanced the current F77 code to incorporate the latest features of Fortran 90 and 95 to conform to the current Fortran standards. Other enhancements involve fine-tuning the code / program for optimization, parallelization and compatibility.

ACKNOWLEDGMENTS

I wish to extend gratitude to James MacLachlan, my supervisor, and Francois Ostiguy, back-up supervisor for their unending help during the course of completing this project. Appreciation also goes to Dave Peterson and Catherine James for their guidance and support during the mentoring sessions. Finally, I would like to thank Dianne Engram, Dr. McCrory and Dr. Davenport and all the members of the SIST committee for giving me another great opportunity to participate at Fermi National Laboratory, Summer Internships in Science and Technology.

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	OPTIONAL CALCULATIONS IN ESME	1
III.	PROJECT DESCRIPTIONS	2
1.	COMPILEATION ON FORTRAN 90 SUN COMPILER	2
2.	REDEFINING ARRAYS POINTERS USING FORTRAN 90 POINTERS	2
3.	REPLACING COMMON BLOCKS WITH MODULE PROGRAMS	3
4.	SUBROUTINE INTERFACE BLOCKS.....	4
5.	COMPILEATION OF ESME ON GFORTRAN AND G95	4
IV.	USING AUTOTOOLS TO GENERATE GNU MAKEFILE FOR ESME	4
V.	CONCLUSIONS.....	6
	LIST OF FIGURES.....	7
	BIBLIOGRAPHY	13
	APPENDIX: BUILDING ESME WITH AUTOCONF, AUTOMAKE TOOLS	14

ESME CODE DEVELOPMENT

I. INTRODUCTION

ESME was written and developed by James MacLachlan of the Accelerator Division / Proton Source at Fermi National Laboratory in FORTRAN77. Initially, it was designed for the Tevatron Antiproton Source¹. ESME is a computer simulation program designed to model those aspects of beam behavior in a synchrotron that are governed by the radio frequency systems (MacLachlan 3). It follows the evolution of a distribution in energy-azimuth coordinates by iterating a map corresponding to the single-particle equation of motion.

ESME is used frequently to assess the efficiency of a given radio frequency beam manipulation or to optimize system parameters (3). Thus, the user needs to specify technical details of the various subsystems and derive various numerical measures or system performance from the particle distribution. ESME is intended for a wide range of detail by separating functions so that only the relevant ones need be considered and by establishing reasonable defaults to reduce the data required for typical cases. ESME is not an acronym, but the title of the J.D Salinger's story "TO Esme with Love and Squalor". The author of this program chose that name because of the girl - heroine in the story.

II. OPTIONAL CALCULATIONS IN ESME

ESME provides three general types of optional calculations based on the properties of the distribution of particles in energy. The most common calculations are those which quantify properties of the distribution so that one can plot them as functions of time (5). The calculations include first and second moments, emittance, Fourier spectrum of beam current, feedback contributions, and evaluation of beam induced voltages from space charge and longitudinal coupling impedance.

III. PROJECT DESCRIPTION

Because ESME is almost 25 years old, a few improvements are needed to make the software more easily portable across many platforms. Several of the enhancements that have been recently made include compilation on the Sun Solaris FORTRAN90 compiler, redefining pointers and arrays using ALLOCATABLE attribute, replacing COMMON blocks with Module programs, and subroutine interface blocks. Configuration scripts for easy code distribution, make script generating, and effective compilation using multiple compiler options have been developed using the GNU Autotools

1. COMPIRATION ON FORTRAN 90 SUN COMPILER

ESME code was previously compiled with the Sun Solaris FORTRAN77 Sun compiler at the time I arrived at Fermilab. My first task was to get a successful compilation of the code on the FORTRAN90 compiler. Major diagnostics were fixed at the initial compilation and the code worked with test data. The source code is divided into three categories: plotting routines, user-written code, and EMSE subroutines. The plotting routines utilize the PG PLOT library to plot various graphs from the program. The user-written subroutines are those routines that are written for specific users based on their requirements and needs. See Figure 1 for more details.

2. REDEFINING ARRAYS POINTERS USING FORTRAN 90 POINTERS

Arrays that were used in ESME program were declared and allocated using the Cray pointer extension to F77. It is a memory-addressing facility that points to or addresses various work areas of an array (Adams 1). Pointers provide a convenient and efficient way for programmers to work conveniently. The pointer is of type integer, and the type of the target can change during program execution. Also, two names are associated with a Cray pointer, the name that refers to the pointer target and the name of the pointer. The memory allocation for a pointee is done through a system subprogram called MALLOC. Another subprogram is called to deallocate the space after the program is done using the space. All the arrays that are utilized by ESME have been redefined

using FORTRAN 90 pointers. With FORTRAN 90 pointers, a pointer is an attribute and the attribute may be given any data object, including an object of user-defined type (1). It is also capable of holding the machine address and the data type. Memory allocation is achieved through the ALLOCATE and DEALLOCATE executable statements. Refer Figure 2, 3 and 4 for how memory allocation is done.

3. REPLACING COMMON BLOCKS WITH MODULE PROGRAMS

ESME acquires its data via FORTRAN77 NAMELIST statements and COMMON blocks. The NAMELIST establishes the name of a collection of objects that can be referenced by the group name in certain input / output statements (Adams, Brainerd, Martin, Smith and Wagener, 144). COMMON blocks are used to store data relating to different program functions or accelerator subsystems (MacLachlan, 8). These blocks are initialized with values or switches to allow the program to proceed on the basis of a few input data. The common blocks allow two or more program units to share the space and values of the variables stored in the space. One of the major problems with common blocks is that the programmer has to replicate the COMMON declaration in each of the sharing program units (HP Documentation). Also, program units may not access the same data if any of the common variables are out of order or have different type or size. As a solution to these issues, Standard FORTRAN77 provided the INCLUDE extension which enables the user to centralize common block definitions in one file. While it solves the problem, it is nonstandard FORTRAN77 and its use is not portable. The best remedy is to utilize one of the new features of Standard FORTRAN 90, the module program. A module is a program unit that facilitates shared access to data and procedures. All shareable variables are declared in the module and any program can reference the entire module via the USE statement. The ONLY clause specifies which module variables are accessible to a particular program unit. See Figure 5 for illustration of module programs.

4. SUBROUTINE INTERFACE BLOCKS

Among the improvements to ESME source code, is the addition of interface blocks to subroutines that make external procedure calls or calls by means of dummy statements. Interface blocks serve to protect and to preserve the variables that are called in the actual procedures. The purpose of the interface blocks is to make the necessary information available to the calling routine so that a procedure reference will be processed correctly. The INTENT attribute is used to specify the intention and purpose of variables for dummy arguments. An INTENT specification could be IN, OUT or INOUT. Refer to Figure 6 for an illustration.

5. COMPIRATION OF ESME WITH GFORTRAN AND G95

The current working version proved fruitful on the Sun compiler with few data set tests; however, I have also compiled ESME with the newest version of GNU GFORTRAN and G95 compiler. Successful compilation was achieved, but because both compilers are still under beta testing and development, some of ESME data sets terminated abruptly apparently because of the current phase of development of the compilers.

IV. USING AUTOTOOLS TO GENERATE GNU MAKEFILE FOR ESME

At present, ESME utilizes a GNUmakefile, a hand-written make script that is used for compiling all the source codes that the program uses. The GNUmakefile is a specification of dependencies between files and how to resolve those dependencies such that an overall goal, known as a target can be reached. This script contains various definitions and specifications for the ESME program. It includes definitions such as: list of source codes (in order of dependency), path to FORTRAN runtime libraries, load flags, compiler flags, and definitions of various FORTRAN compilers and etc. On the other hand, the GNU software foundation organization group has developed a set of tools called Autotools for performing major tasks of compiling source codes, software distribution, and all other system checks

before software is installed. The set of Autotools are Autoconf, Automake, and Libtools. In this report, I utilized the first two of the Autotoolset to develop a minimal configuration script that compiles the ESME code successfully and generates a GNUmakefile that behaves just like the hand-written one. The configuration script can be used with other “--with-PACKAGE” options for various compilers where the PACKAGE represents the compiler name or software name.

Autoconf is a tool for generating shell scripts which systematically configure software source code packages to adapt to many kinds of UNIX-like systems (MacKenzie 1). The configuration scripts produced by Autoconf are independent of the systems on which they are run; therefore, their users need not to have them installed. Autoconf requires GNU M4 in order to generate the configure script. M4 is a general purpose tool suitable for all kinds of text processing applications (Vaughan). Its application is as a front-end for a compiler. M4 reads text from the input and writes processed text to the output. It includes a set of pre-defined macros that make it substantially more useful. These macros perform functions such as arithmetic, conditional expansion, string manipulation and running external shell commands.

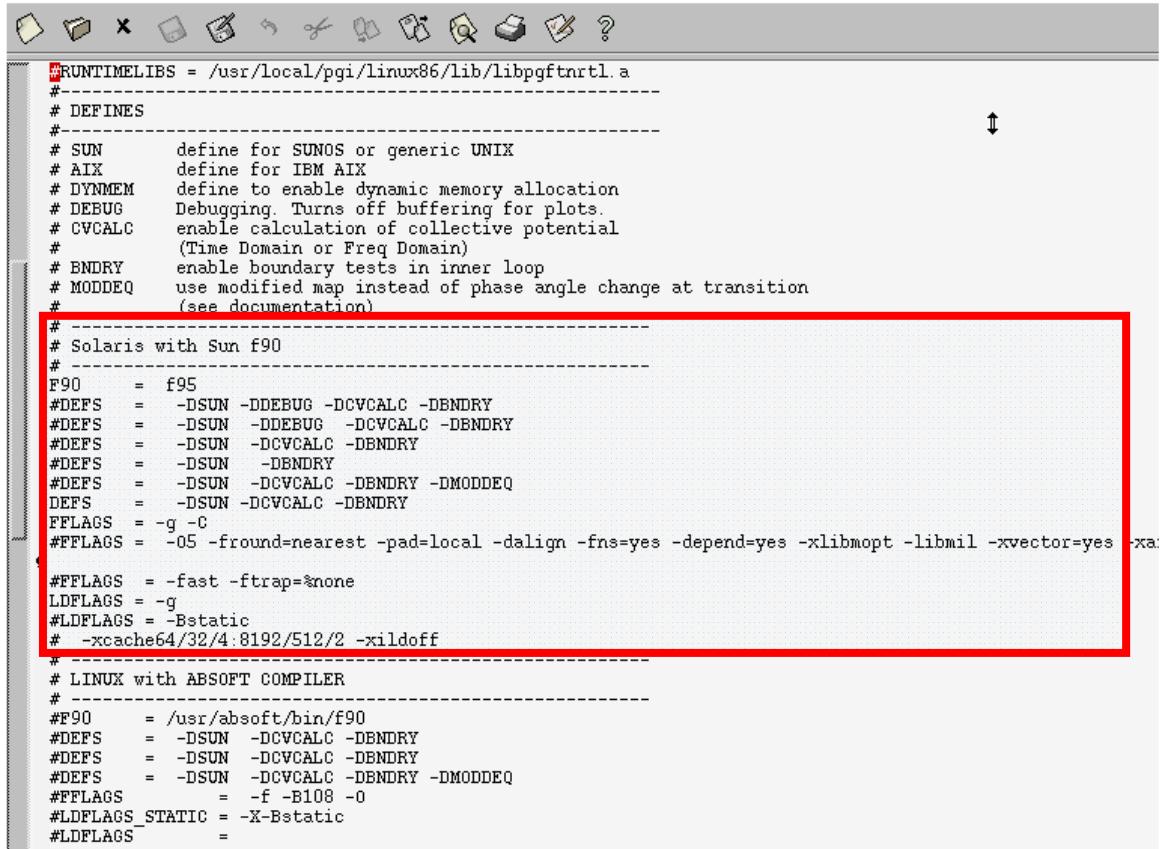
Automake is a tool for automatically generating *GNUmakefile.ins* from files called *GNUmakefile.am* (GNU SF). Each *GNUmakefile.am* is a series of make variable definitions (make macros or Autoconf macros) with rules being thrown in occasionally. Automake requires Perl in order to generate the Makefile.ins; however, the distributions created do not require Perl in order to be built. To write the configuration script for ESME, two files were used: configure.ac and *GNUmakefile.am* in the top directory. All other sub directories must have its own *GNUmakefile.am* with various targets and destinations defined. See appendix A for a step by step to use Autoconf and Automake for ESME project.

V. CONCLUSIONS

Incorporation of new features of Standard FORTRAN 90 into the current ESME code was a success.

Also, three successful compilations were achieved. The configuration script generated using the GNU Autotools – Automake and Autoconf will ease the burden of porting the program to different operating systems and replace archaic make files for users worldwide.

LIST OF FIGURES



The figure shows a terminal window with a grey header bar containing various icons. The main area of the terminal contains a hand-written makescript. A red rectangular box highlights a section of the script. The script includes definitions for runtime libraries, defines, and compiler flags for different platforms (SUN, AIX, DYNMEM, DEBUG, CVCALC, BNDRY, MODDEQ) and compilers (f90, f95). It also includes sections for Solaris with Sun f90 and Linux with ABSOFT COMPILER.

```
RUNTIMELIBS = /usr/local/pgi/linux86/lib/libpgftnrtl.a
#
# DEFINES
#
# SUN      define for SUNOS or generic UNIX
# AIX      define for IBM AIX
# DYNMEM   define to enable dynamic memory allocation
# DEBUG    Debugging. Turns off buffering for plots.
# CVCALC   enable calculation of collective potential
#          (Time Domain or Freq Domain)
# BNDRY    enable boundary tests in inner loop
# MODDEQ   use modified map instead of phase angle change at transition
#          (see documentation)
#
# Solaris with Sun f90
#
F90      = f95
#DEFS    = -DSUN -DDEBUG -DCVCALC -DBNDRY
#DEFS    = -DSUN -DDEBUG -DCVCALC -DBNDRY
#DEFS    = -DSUN -DCVCALC -DBNDRY
#DEFS    = -DSUN -DBNDRY
#DEFS    = -DSUN -DCVCALC -DBNDRY -DMODDEQ
DEFS     = -DSUN -DCVCALC -DBNDRY
FFLAGS   = -g -O
#FFLAGS  = -O5 -fround=nearest -pad=local -dalign -fns=yes -depend=yes -xlibmopt -libmil -xvector=yes -xa:
#FFLAGS  = -fast -ftrap=none
LDFLAGS  = -g
#LDFLAGS = -Bstatic
# -xcache64/32/4:8192/512/2 -xildoff
#
# LINUX with ABSOFT COMPILER
#
#F90      = /usr/absoft/bin/f90
#DEFS    = -DSUN -DCVCALC -DBNDRY
#DEFS    = -DSUN -DCVCALC -DBNDRY
#DEFS    = -DSUN -DCVCALC -DBNDRY -DMODDEQ
#FFLAGS   = -f -B108 -O
#LDFLAGS_STATIC = -X-Bstatic
#LDFLAGS  =

```

Figure 1 shows a hand-written makescript with various Sun Compiler flags settings

The screenshot shows a web browser window with the URL <http://www-ap.fnal.gov/users/jmaclach/include/dynmem.inc>. The page content is a C header file named `dynmem.inc`. The code defines several pointer types using Cray's pointer extension syntax. A red box highlights a section of the code that declares pointers to arrays of `REAL` type.

```
ifndef _DYNMEM_INC
#define _DYNMEM_INC
    DOUBLE PRECISION PHASETH(0:NPHASE), PHASEEE(0:NPHASE)
    POINTER (PTRPHASETH, PHASETH)
    POINTER (PTRPHASEEE, PHASEEE)

    REAL SPHASETH(NPHASE), SPHASEEE(NPHASE)
    POINTER (PTRSPHASETH, SPHASETH)
    POINTER (PTRSPHASEEE, SPHASEEE)

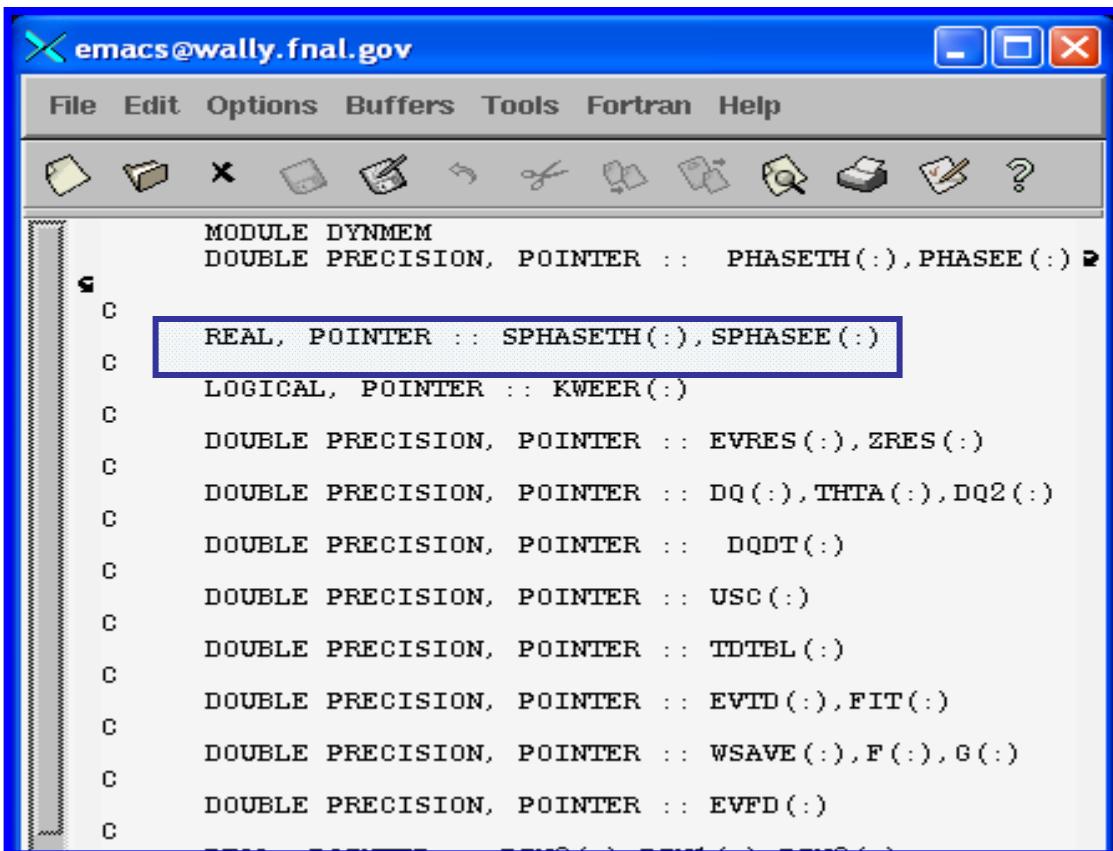
    LOGICAL KWEER(NPHASE)
    POINTER (PTRKWEER, KWEER)

    DOUBLE PRECISION EVRES(MAXCVB), ZRES(MAXCVB)
    POINTER (PTREVRES, EVRES)
    POINTER (PTRZRES, ZRES)

    DOUBLE PRECISION DQ(MAXCVB), THTA(MAXCVB), DQ2(MAXCVB)
    POINTER (PTRDQ, DQ)
    POINTER (PTRTHTA, THTA)
    POINTER (PTRDQ2, DQ2)

    DOUBLE PRECISION DQDT(MAXCVB)
```

Figure 2 shows the use of Cray Pointer extension to declare and point to an array

A screenshot of an Emacs window titled "emacs@wally.fnal.gov". The window has a blue header bar with icons for minimize, maximize, and close. Below the header is a menu bar with "File", "Edit", "Options", "Buffers", "Tools", "Fortran", and "Help". A toolbar follows with icons for file operations like open, save, and search. The main buffer contains Fortran code. A specific line of code is highlighted with a blue rectangular box:

```
MODULE DYNMEM
  DOUBLE PRECISION, POINTER :: PHASETH(:, :), PHASEE(:, :)
  C
  C      REAL, POINTER :: SPHASETH(:, :), SPHASEE(:, :)
  C      LOGICAL, POINTER :: KWEER(:, :)
  C      DOUBLE PRECISION, POINTER :: EVRES(:, :), ZRES(:, :)
  C      DOUBLE PRECISION, POINTER :: DQ(:, :), THTA(:, :), DQ2(:, :)
  C      DOUBLE PRECISION, POINTER :: DQDT(:, :)
  C      DOUBLE PRECISION, POINTER :: USC(:, :)
  C      DOUBLE PRECISION, POINTER :: TDTBL(:, :)
  C      DOUBLE PRECISION, POINTER :: EVTD(:, :), FIT(:, :)
  C      DOUBLE PRECISION, POINTER :: WSAVE(:, :), F(:, :), G(:, :)
  C      DOUBLE PRECISION, POINTER :: EVFD(:, :)
```

Figure 3 shows Fortran 90 standard method of declaring array pointers.

The screenshot shows an Emacs window titled "emacs@wally.fnal.gov" displaying a Fortran program. The window has a menu bar with File, Edit, Options, Buffers, Tools, Fortran, and Help. A toolbar with various icons is located above the code area. The code itself is as follows:

```
4      IFFT, KIFFT
      ENDIF
c
      ALLOCATE (PHASETH(0:KNPHASE), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate PHASETH'
      DO 100, I=0,KNPHASE
         PHASETH(I) = ZERO
100   CONTINUE
      ALLOCATE (PHASEEE(0:KNPHASE), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate PHASEEE'
      DO 200, I=0,KNPHASE
         PHASEEE(I) = ZERO
200   CONTINUE
      ALLOCATE (SPHASETH(KNPHASE), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate SPHASETH'
      ALLOCATE (SPHASEEE(KNPHASE), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate SPHASEEE'
      ALLOCATE (KWEER(KNPHASE), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate KWEER'
      ALLOCATE (DQ(KMAXCVB), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate DQ'
      ALLOCATE (THTA(KMAXCVB), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate THTA'
      ALLOCATE (DQ2(KMAXCVB), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate DQ2'
      ALLOCATE (EVRES(KMAXCVB), STAT=ITST)
      IF (ITST.NE.0) STOP 'Fail allocate EVRES'
      ALLOCATE (DQDT(KMAXCVB), STAT=ITST)
--:-- memalloc.F          (Fortran)--L43--30%
```

Figure 4 shows how the arrays are allocated memory space in FORTRAN 90 pointers

```
emacs@wally.fnal.gov
File Edit Options Buffers Tools Fortran Help
MODULE FEEDS
C PARAMETERS DEFINING FEEDBACK LOOPS (CURRENTLY PHASE & VOL)
C
C USE PARAMETERS
C
C     DOUBLE PRECISION :: DLIMIT=5.7296D0, PLIMIT, FBFAC=1.D
C     1 VLIMIT=1.D-1, VFBFCTR=1.D+0, PSIADD, DAMPL, SUMW, CUR
C     2 PASTH, THRMSL, PRGRMSL, W(IFBPRS), BLTABL1(LTABL),
C     3 BLTABL2(LTABL), BLTABL3(LTABL), UTBL(LTABL), FBPRS(I
C
C     INTEGER :: NTUAVG=1, NTURES=1, IFTB=0, NTL, NCBEG, NCEND,
C     1 NOBEG, NOEND, NTABBL
C
C     LOGICAL :: PHFBON=.FALSE., VFBBON=.FALSE., USEWT=.FALSE..
C
C     TYPE XFEEDS
C     DOUBLE PRECISION DLIMIT, PLIMIT, FBFAC, VLIMIT, VFBFCTR, P
C     1 DAMPL, SUMW, CURRH, PASTH, THRMSL, PRGRMSL, BLTABL1,
C     2 BLTABL2, BLTABL3, UTBL, FBPRS, W
C     INTEGER NTUAVG, NTURES, IFTB, NTL, NCBEG, NCEND, NOBEG, NOEND
C     LOGICAL PHFBON, VFBBON, USEWT, EXFG
C     END TYPE XFEEDS
C     TYPE (XFEEDS) SFEEDS
C
C     END MODULE FEEDS
```

Figure 5 illustrates common blocks replaced with module programs

```
SUBROUTINE BEAMSC
C
USE PARAMETERS
USE RINGP, ONLY : THRNG
USE FOURIR, ONLY : NBINFFT, BW, NF, NNF
USE SPCHARGE, ONLY : NBINSC, NBRES, NZ, NR, ZTABL, TDELTA, ARTP, MSC,
1   TCHGON, FDON, FDSCON, SCON, QREZON, TDON, RESTBL, BWSC, IRLLEN, NT
USE CURRENT, ONLY : DELEIMP, P0
USE IO, ONLY : KMAXCVB, KIFFT, KOUTUNIT, KVERBOSE, KINUNIT
USE DYNMEM, ONLY : TDtbl

C
IMPLICIT DOUBLE PRECISION (A-H, O-Z)

C
INTERFACE
SUBROUTINE DELESC(DOKICK, SCSTEP)
IMPLICIT DOUBLE PRECISION (A-H, O-Z)
LOGICAL, INTENT (IN) :: DOKICK
INTENT(IN) :: SCSTEP
END SUBROUTINE DELESC
END INTERFACE

C
INTERFACE
SUBROUTINE HIQRES(DOKICK, SCSTEP)
IMPLICIT DOUBLE PRECISION (A-H, O-Z)
LOGICAL DOKICK
END SUBROUTINE HIQRES
END INTERFACE

C
INTERFACE
SUBROUTINE FDVCOLL(DOKICK, SCSTEP)
IMPLICIT DOUBLE PRECISION (A-H, O-Z)
LOGICAL DOKICK, NEEDIT
END SUBROUTINE FDVCOLL
END INTERFACE

C
--:-- beamsc.F          (Fortran)--L1--Top--
Loading fortran...done
```

Figure 6 illustrates the use of Interface blocks with the INTENT attribute

BIBLIOGRAPHY

Adams, Brainerd, et al., Fortran 95 Handbook, complete ISO / ANSI reference Cambridge: The MIT Press, 1997.

Adams, Jeanne. Comparing pointers in Cray Fortran and Fortran 90, 27 Aug. 2005
<<http://www.scd.edu/tcg/consweb/Fortran90/scnpoint.html>>

Gelato, Bierman, and Grosse. User Notes on Fortran Programming (UNFP): Dynamic memory allocation and pointers. 28 Aug. 2005 <<http://www.ibiblio.org/pub/languages/fortran/ch2-16.html>>

MacLachlan, James; “Users Guide to ESME 2005” January 2005.

MacKenzie, David. Autoconf: Creating Automatic Configuration Scripts, 24 Aug. 2005
<<http://www.amath.washington.edu/~lf/tutorials/autoconf/autoconf/autoconf.html>>

Vaughan, Elliston, et al., GNU Autoconf, Automake, and Libtool, Peachpit Press, 2001
Fortran 90 Programmer’s Guide: HP 9000 Computers. 15 July 2005.

<<http://docs.hp.com/en/B3909-90002/index.html>>

APPENDIX: BUILDING ESME WITH AUTOCONF, AUTOMAKE TOOLS

This tutorial is intended to show how the GNU Autotools was used to generate a configuration script for the ESME simulation program. Users of other languages can also adapt it to their project. The way things are done here is slightly different because of the language used – FORTRAN. The material described here is a step by step procedure / tutorial of I how utilized Autoconf & Automake to build a configuration script (Makefile or GNUMakefile).

For more information on ESME or how this configuration script works, please see <http://www-ap.fnal.gov/ESME> or forward your inquires to James MacLachlan at Fermi National Laboratory. The steps described here were adapted partly from sourceforge.net and from GNU software foundation website.

Contents

- I. Preparing your directory structure
 - Creating configure.ac
 - Creating Makefile.am
- II. Generating Output Files
- III. Building and Installing the Project.
- IV. Maintaining Input Files
- V. Other useful resources

STEP 1: Preparing project directory and sub directories.

1. Create a directory called **auto_esme**, for example; his directory will be the top directory for the EMSE project for a new build.
2. Create appropriate sub-directories, e.g. *src*, *modules*, etc. Move the respective source codes in the appropriate sub-directories.

STEP 2: Creating minimal files needed to start script generation process.

Steps that I described here can be done in another way if the user knows more about m4 macros. Read more on m4 macros [here](#).

1. We need to create 2 files in the top directory (auto_esme). The first file is called *configure.ac*. See macro descriptions after *configure.ac* script.

'**configure.ac**'

```
AC_INIT(src/beams.F)
##AM_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(esmF95, 1.0, no-define)

AC_ARG_WITH([g95],AC_HELP_STRING([--with-g95],[use GNU g95]), if test
$withval=yes; then F77=g95; else F77=$withval; fi; fortran_g95=1)
```

```

AC_ARG_WITH([gfortran],AC_HELP_STRING([--with-gfortran],[use GNU
gfortran]), if test $withval=yes; then F77=gfortran; else
F77=$withval; fi; fortran_gfortran=1)
AC_ARG_WITH(pgf77,AC_HELP_STRING([--with-pgf77],[use PGI
fortran]),F77=pgf77;fortran_pgi=1)
AC_ARG_WITH(absoft,AC_HELP_STRING([--with-absoft],[use Absoft
fortran, using f77 or specify]),F77=f77;fortran_absoft=1)
AC_ARG_WITH(sun,AC_HELP_STRING([--with-sun],[use SUN fortran, using
f77 or specify]),F77=f77;fortran_sun=1)
AC_ARG_WITH(g77,AC_HELP_STRING([--with-g77],[use GNU
fortran]),F77=g77)
AC_PROG_F77(f95 f90 f77 pgf77 g77 g95 gfortran)
AC_PROG_CPP

# use rindex.F if using GNU compiler
AM_CONDITIONAL(GNU_RINDEX,test ${G77})

AC_DEFINE(SUN,1,sun)
##AC_ARG_ENABLE(dynamic-memory,AC_HELP_STRING([--disable-dynamic-
memory],[not supported with g77]),,if test "x${G77}" = "xyes"; then
AC_DEFINE(G77,1,g77) else AC_DEFINE(DYNMEM,1,dynamic memory) fi)
AC_ARG_ENABLE(debug,AC_HELP_STRING([--enable-debug],[Debugging.
Turns off buffering for plots.]),AC_DEFINE(DBG,1,debug))
AC_ARG_ENABLE(cvcalc,AC_HELP_STRING([--enable-cvcalc],[disable
calculation of collective potential]),,AC_DEFINE(CVCALC,1,cvcalc))
AC_ARG_ENABLE(boundary,AC_HELP_STRING([--enable-boundary],[disable
boundary tests in inner loop]),,AC_DEFINE(BNDRY,1,boundary))
AC_ARG_ENABLE(moddeq,AC_HELP_STRING([--disable-moddeq],[use modified
map instead of phase angle change at
transition]),AC_DEFINE(MODDEQ,1,moddeq))

AC_LIB_PGPLOT
AC_LIB_X
AC_LIB_RUNTIME
AC_OUTPUT(GNUmakefile src/GNUmakefile)

```

The configure.ac script does the followings:

- The `AC_INIT` macro performs essential initialization for the generated configure script. It takes a filename as an argument from the source directory to ensure that the source directory has been correctly specified.
- The `AM_CONFIG_HEADER` macro is used to request the use of a configuration header. It is commented out in this case because I wanted to see all the flags i.e. `-DSUN`, `-DVCALC` etc. when I run the ‘`gmake`’ or ‘`make`’ command.
- The `AM_INIT_AUTOMAKE` performs some further initializations that are related to the fact that we are using ‘`automake`’. If you are writing your ‘`Makefile.in`’ by hand, then you don’t need to call this command.
- The `AC_ARG_WITH` macro is used to give other users with external software package an option to compile the program with that package with use of ‘`--with-PACKAGE`’ where PACKAGE is the name of the software e.g. `gfortran` or `g95`.
- The `AC_PROG_F77` checks to see which FORTRAN compiler is available, it uses the first one that it encounters unless otherwise specified.
- The `AC_ARG_ENABLE` enables the user to define another command line options. In this case, it is used to enable `cvcalc`, `bndry`, and `debug`.

- The `AC_LIB_PLOT`, `AC_LIB_X`, and `AC_LIB_RUNTIME` macros are defined in the optional input file called `acinclude.m4`. They are defined using the `AC_DEFUN` macro for the configure script to do various test such as guessing the location of library files or compiler locations.
- The `AC_OUTPUT` macro must be invoked at the end of the `configure.ac` to create the Makefiles.

'GNUmakefile.am'

I provided a `GNUmakefile.am` file for each directory in your source tree. `GNUmakefile.am` for the top-level directory is simple. It looks like this:

```
SUBDIRS = src
```

The `SUBDIRS` variable is used to list the subdirectories that must be built.

Next, I created another `GNUmakefile.am` in the `src` directory. This `GNUmakefile.am` looks a bit different in that it lists all the source files in order of dependencies.

```
#####
## Written by Ayodele T. Onibokun, SIST 2005
## with special assistance of Francois Ostiguy, FNAL
## ESME author(s): James MacLachlan, Francois Ostiguy, FNAL
## Fermi National Laboratory
#####

SUFFIXES: .c .f .f90 .F .o .mod

bin_PROGRAMS=esmF95
esmF95_SOURCES= $(esmF95_modules) $(esmF95_src) $(esmF95_GRAPHSRC) \
                 $(esmF95_SHAZSRC)

# Source files for new PGPlot graphics
#
esmF95_GRAPHSRC = drawinit.F esinit.F esmain.F esquit.F \
                  exclbnd.F grafset.F history.F mrplt.F \
                  phplt.F pltcntur.F pltenergy.F pltfdvc.F \
                  pltfou.F pltphase.F pltresn.F pltspchg.F \
                  pltdbf.F plttdvc.F pltthta.F pltwav.F \
                  pltztab.F select.F

# User-written subroutines

esmF95_SHAZSRC = gmoments.F pltfrqnc.F pltshaz5.F shazam.F \
                  shazam1.F shazam2.F shazam3.F shazam4.F \
                  shazam5.F shazam6.F shazam7.F shazam8.F \
                  shazam9.F

# ESME 2005 Source Codes

esmF95_src = beamsc.F beedot.F bernst.F bfunct.F \
              bvolt.F bnch.F buckit.F \
```

```

chgdist.F contour.F cycprog.F dbfun.F \
dbvolt.F delesc.F display.F dlaran.F \
dtabout.F drfv.F eloss.F engfmt.F \
evhbfix.F evsbfix.F fampout.F famphst.F \
fampwrt.F fdvcoll.F fft.F ffti.F \
fftset.F fillbe.F fillbfg.F fillbg.F \
fillbp.F fillbpr.F fillbr.F fillbu.F \
fillfix.F foline.F flowprg.F fourr.F \
freqnc.F gammas.F gcd.F get.F \
getascii.F getbin.F getdat.F glabl.F \
hires.F histwr.F intpgam.F inverf.F \
karve.F lentrue.F linbkt.F loop.F \
lowlvl.F match.F mbfcotr.F memalloc.F \
memfree.F moments.F mrinit.F mrsave.F \
nrbis.F outlb.F outside.F phases.F \
phasflo.F phfeed.F point.F popul8.F \
prefix.F prntout.F refcont.F rfprog.F \
ringpar.F rmoments.F rootf.F rtangl.F \
rfv.F savascii.F save.F savbin.F \
septrix.F sliph.F spline.F strays.F \
synch.F \
tablout.F tdvcoll.F topbtm.F trap.F \
volts.F vfeed.F wght.F ztabout.F \
ztabspl.F

# ESME 2005 Modules (formerly known as includes)

esmF95_modules = \
./modules/parameters.f \
./modules/bucket.f      ./modules/dynmem.f      ./modules/bktsup.f \
./modules/blankcom.f   ./modules/bunch.f \ \
./modules/const.f       ./modules/current.f    ./modules/curves.f \
./modules/cyclp.f      ./modules/feeds.f \ \
./modules/flowp.f       ./modules/fourir.f    ./modules/grafix.f \
./modules/histcom.f \ \
./modules/io.f           ./modules/mtnrange.f   ./modules/pgpltcom.f \
./modules/plt.f          ./modules/poplate.f \ \
./modules/random.f       ./modules/rfp.f        ./modules/ringp.f \
./modules/shaz2.f         ./modules/shaz5.f \ \
./modules/shaz7.f         ./modules/spares.f   ./modules/spcharge.f \
./modules/times.f

esmF95_LDADD   = @X11LIBS@ @RUNTIMELIBS@ @PGPLOTLIB@ @LOCALIBS@
esmF95_LDFLAGS = -R /usr/local/ap/X11R6/lib

AM_CPPFLAGS = -I./modules

Distclean-local:
    rm -f *.mod

```

- The *bin_PROGRAMS* variable specifies that we want a program called *esmF95* to be built and installed in the bin directory when *gmake* is run.
- The *esmF95_SOURCES* variable specifies the source files used to build the esmF95 target. Also, note that the *SOURCES* variable for a target is prefixed by the name of the target, in this case *esmF95*.

- The *SUFFIXES* specifies how to build the corresponding *.o*, *.mod* files from the source files given i.e. *.f*, *.F* etc. For a simple Hello program, this line is probably not needed.

STEP 3: Generating Output Files

I had to generate the required output files from the two input files *configure.ac* and *Makefile.am*. The first step is to collect all the macro invocations in *configure.ac* that Autoconf will need to build the configure script. This is done with the following command:

```
$ aclocal
```

NOTE: You must still be in the top directory auto_esme

The aclocal command then generates a file called *aclocal.m4* in the current directory

Next, *aclocal.m4* and *configure.ac* file will be used to generate the configuration (configure) script. The command is:

```
$ autoconf
```

There are a few files that the GNU standard says must be present in the top-level directory, and if not found Automake will raise an objection. The following will generate these missing files:

```
$ touch AUTHORS NEWS README ChangeLog
```

Next, it is time to generate the *Makefile.ins*, the following command does it:

```
$ automake -a
```

The '*-a*' copies some boilerplate files from your Automake installation into the current directory.

The content of the top directory should look something like this:

AUTHORS	GNUmakefile	INSTALL	acinclude.m4	config.log
configure.ac	modules	COPYING	GNUmakefile.am	NEWS
aclocal.m4	config.status	install-sh	src	ChangeLog
GNUmakefile.in	autom4te.cache	configure	missing	README

Step 3: Installing the Project

At this point, the project is ready to be installed (compiled). The command is:

```
$ gmake
```

If you make changes to any of the source files, you have to issue '*gmake*' or *gmake distclean*' then '*gmake*' again to re-compile the changes.

Run the program with test data and see that it works just like it would if it was manually done by hand without the *configure* script. Change directory into the *src* and look for an executable file called *esmF95*. Issue the usual commands:

```
$ esmF95 -i -f [datafile.dat] {for example merging.dat (no square braces)}
```

Step 4: Building the Project for the Users

At this point, the project is ready to be packaged in a tarball and be distributed. To avoid having mixed the project files with the configure files in the top directory, I recommend to make a sub directory called **BUILD** under **auto_esme**. While at the top directory, do a:

```
$ gmake distclean
```

Then issue

```
$ autoconf
```

And

```
$ automake
```

at the top directory.

Next, change directory to **BUILD** and do

```
$ ./configure
```

Then issue

```
'gmake distcheck' to generate the tarball file and the distribution is  
ready to go.
```

If everything goes as described, you should see something like this:

```
=====  
esmF95-1.0 archives ready for distribution:  
esmF95-1.0.tar.gz
```

Installing the project at the Users' end

A user just has to unpack the tarball and run the following commands:

```
$ tar -zxvf [filename with all extensions]  
$ ./configure --prefix=/some_directory  
$ gmake
```

Maintaining Input Files

Everytime there is a change to the GNU Autotools input files in your package, you must regenerate the output files. If you are building your package you will need to re-run `configure` to regenerate the Makefile's. Many project maintainers put the necessary commands to do this into a script called `autogen.sh` and run this script whenever the output files need to be regenerated.

Create a text file called `autogen.sh` in the top-level directory and make sure you change its file mode to make it executable. Add the following commands to the file and save it:

```
#!/bin/sh  
  
aclocal\  
&&automake - a\  
&& autoconf
```

Now you can easily run the following commands to update your project's output files, and rebuild the project:

```
$ ./autogen.sh  
$ ./configure --prefix=/some_directory  
$ gmake
```

Other useful resources

The procedure that I described should get you started. Obviously, you'll need to learn more about Autotools. Here are some useful URLs:

- [GNU Autoconf, Automake, and Libtool](#)
- [GNU's automake and autoconf manuals](#)
- [Learning the GNU development tools](#)

Documentation adapted from various sources such as GNU main page, Red Hat sources.

Written by Ayodele Onibokun 09/14/2005