NCR CENTURY SERIES OBJECTIVES

The NCR Century Series of electronic data processing systems is the result of several years of intensive research and planning.

With the release of the NCR Century Series of EDP systems, NCR achieved the following design goals:

- Broad Market Range -- The NCR Century Series consists of several types of processors and peripheral units. The possible system configurations cover a very wide performance range.

- Adaptability -- The great variety of peripheral units for the NCR Century Series makes it possible to adapt NCR Century Systems for efficient use in practically any type of business.

- High Performance and Low Cost -- Three major factors contribute to the high performance and low cost of the NCR Century Systems:

  - Extensive use of software

  - Modern hardware design

  - Modern production techniques

- Information Interchange -- The NCR Century Series uses as its basic unit of information an 8-bit word (byte), which is standard in the EDP industry. The use of this 8-bit word permits easy information interchange and communication between systems of different types and makes.

- Ease of Programming -- NCR supplies with each NCR Century System a complete set of software designed to facilitate the user's programming task. Program compilers (NEAT/3, COBOL, and FORTRAN) permit the writing of programs in problem-related languages. Other software performs system-related functions, such as copying files, etc.

  NCR also supplies complete and flexible programs for many standard applications in various fields.

- Program Compatibility -- As a user's requirements grow, he may acquire a more powerful member of the NCR Century Series without having to rewrite his existing programs. The user may simply recompile his existing source programs to modify them for the most efficient use on the new system configuration.

INTRODUCTION TO NCR CENTURY SERIES HARDWARE

The purpose of this publication is to acquaint the reader with the design philosophy and the basic features of the NCR Century Series hardware.

The NCR Century Series of information processing systems is a completely new line of processors, peripherals, and communication equipment designed to meet the demands of a wide range of customer needs.

DESIGN FEATURES

Common Trunk

All units in the NCR Century Series are designed with common input/output characteristics. This permits the use of a common trunk to connect different types of peripheral units to the processor.

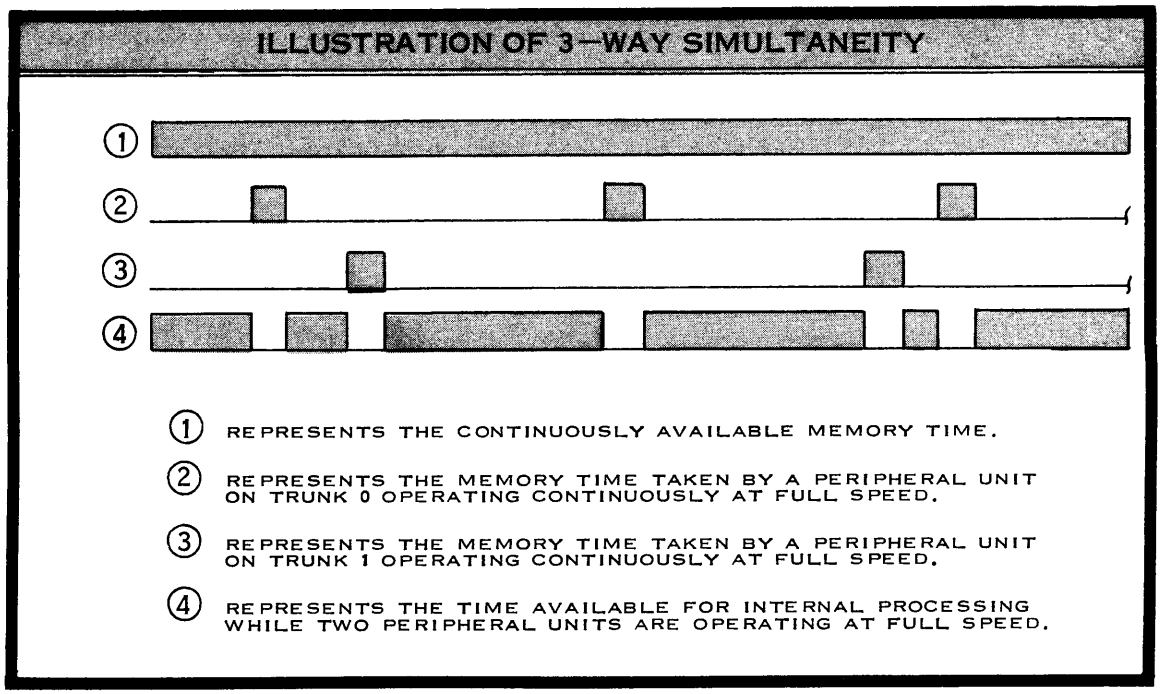
Following are three advantages of the common trunk design:

- Economy -- The NCR Century Systems eliminate excessive input/output hardware.
- Programming convenience -- The processor handles all input/output functions in a similar manner.
- Flexibility -- Present or future peripheral units are easily added to an existing NCR Century System.

Simultaneity

The short cycle time (high speed) of the NCR Century memories permits the simultaneous occurrence of several processing functions. All memory cycles not needed by input/output functions, which have priority, are available to the current program in the processor.

The number of possible simultaneous functions depends on the number of input/output trunks on the processor. For example, the NCR Century 100 with two input/output trunks may perform three functions simultaneously: the current program in the processor may be progressing while data is input from or output to two different peripheral units.



The simultaneous occurrence of several functions in the NCR Century 100 increases the system's performance by reducing program running time.

Upward Program Compatibility

Hardware and software design permits programs written for a lower member of the NCR Century Series to be used on a higher member of the same series. This program compatibility is made possible by the use of disc memories on all systems and by the identical data structure throughout the entire Century Series.

Program compatibility offers the user the following advantages:

- Economy -- The user need only acquire a system to satisfy his needs for the near future. If the user outgrows his present system and moves up to a higher member of the NCR Century Series, the expense of converting his existing programs in minimal.

- Speed of conversion -- No time loss due to reprogramming and debugging occurs when a user acquires a more powerful member of the Century Series.

Flexibility

NCR offers a variety of peripheral units for the various processors in the NCR Century Series. Many similar peripheral units are available with different levels of performance. This variety of units economically fulfills the requirements of practically any application.

If a user's workload exceeds the capacity of his present system, he may get additional units for his NCR Century System, or he may exchange one or several of his present units for like units with a higher performance level.

Integrated Peripheral Units

For greater economy and operator convenience, certain peripheral units in the NCR Century Systems share portions of the cabinet, internal logic circuits, and power supplies with the processor. These peripheral units, referred to as integrated (or system) peripheral units, occupy dedicated positions on the communications (common) trunks. The integrated peripheral units operate in the same manner as other freestanding peripheral units.

The NCR Century 100 base system consists of a processor with memory and the following integrated peripherals:

- Dual spindle disc file.

- Punched card reader or punched tape reader.

- Line printer.

Ease of Operation

The NCR Century Series of computers is designed for convenient and efficient operation. The controls and indicators on all units are located at a com-

fortable height. Any NCR Century System may be arranged so that the operator need only take a few steps to service any peripheral unit. All peripheral units, including the disc units, are designed for easy loading and unloading of file media.

Ease of Installation

In the preparation of a computer site, the user must give consideration to the system's requirements of electrical power, air conditioning, placement of cables, and space.

The lower power consumption and wide temperature tolerance of the micro-electronic circuits in the NCR Century Series reduce the air conditioning requirements for the site.

The design of the units' frames and cabinets eliminates the need for a raised floor at the site. The bases of the adjoining units in the NCR Century Systems house all the connecting cables.

Reliability and Maintainability

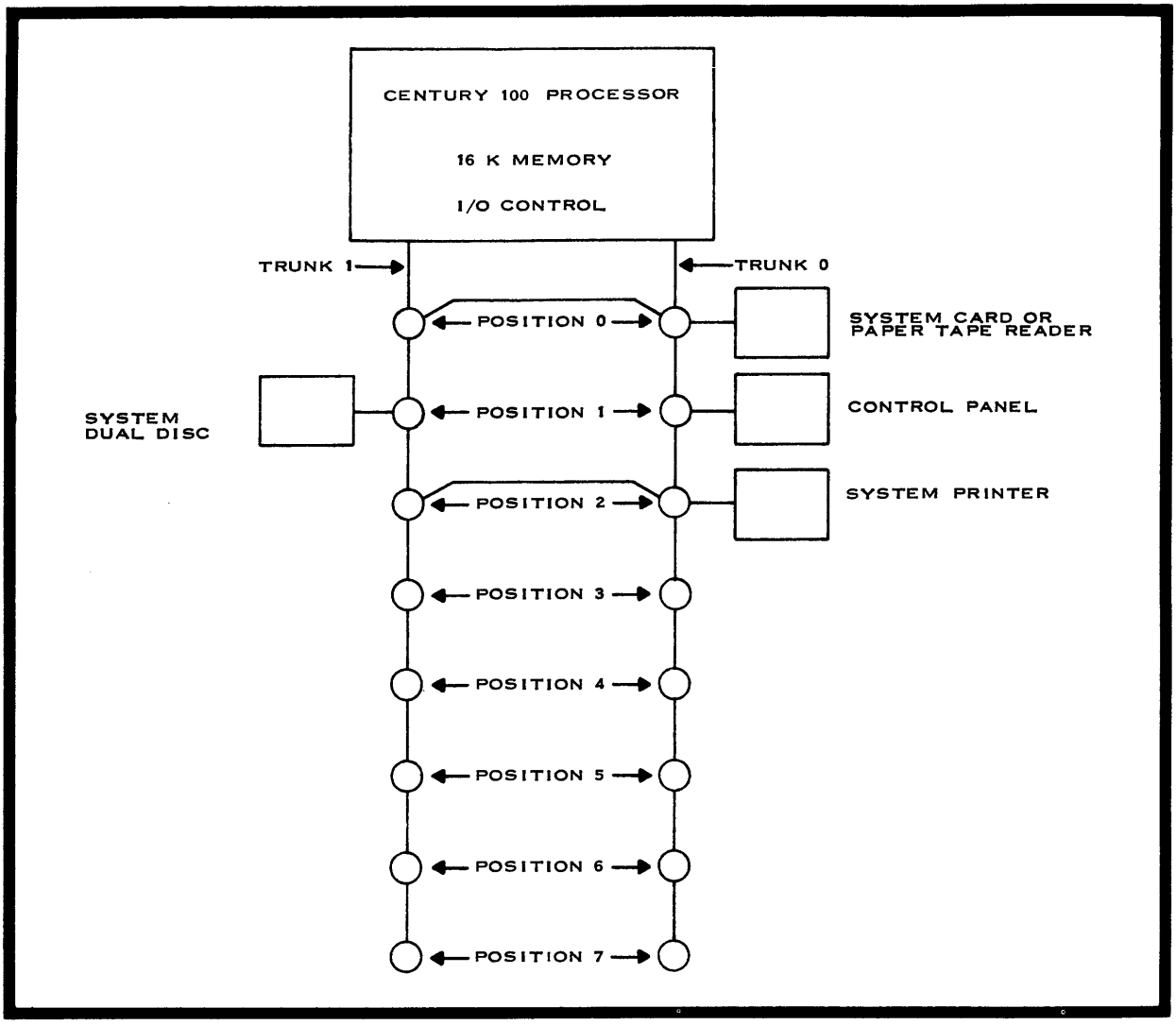
The standardization of circuit boards and the extensive use of integrated circuits give NCR Century Systems a high degree of operating reliability at low cost.

The design of the NCR Century System also permits easy and efficient maintenance of the system. All parts are easily accessible, and the repeated use of standard circuit boards throughout the system reduces the number of service parts required to service the system. This results in reduced maintenance cost.

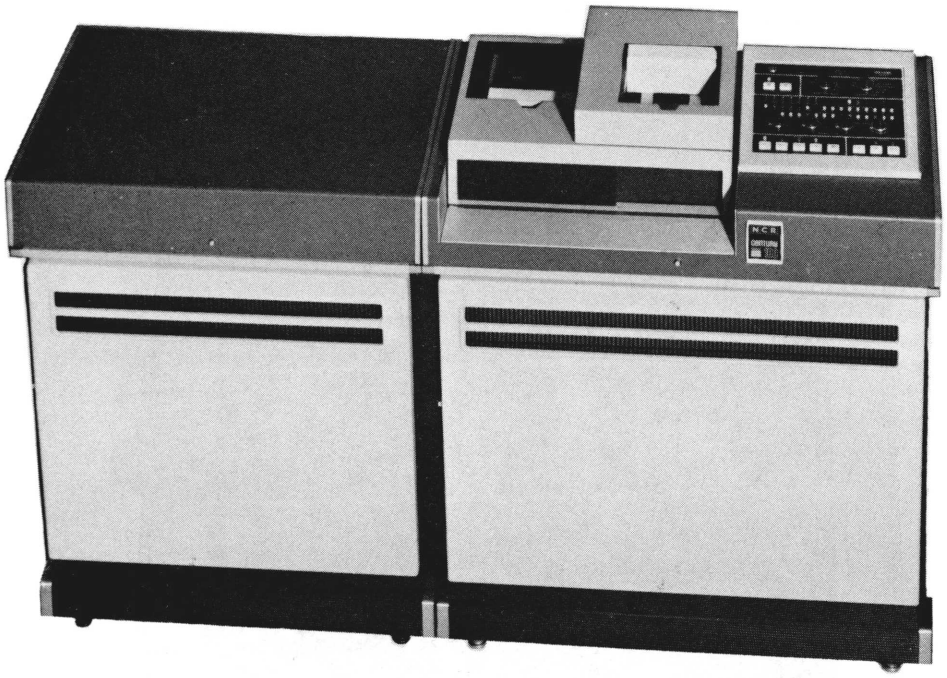
NCR CENTURY 100 BASE SYSTEM

The system shown below represents the minimum system capable of complete and efficient file processing. The units in the system are a processor with integrated memory and operator's control panel, a card reader, a high speed printer, and a dual disc unit. (The user has a choice of either a punched card reader or a punched paper tape reader.)

The following diagram indicates the trunk and trunk position to which each integrated peripheral unit in the base system is assigned. Additional peripheral units may be connected to positions 3 through 7 on both trunks.



NCR Century 100 Central Processor



The NCR Century 100 processor is a general purpose processor with a memory of 16,384 characters.

The hardware in the NCR Century 100 recognizes 19 commands. These commands have a length of either four or eight 8-bit characters. All internal operations in the NCR Century 100 are carried out on a one-character-at-a-time basis. Each character in memory is separately addressable. The hardware considers all numerical data as unsigned, unpacked, and either decimal or binary.

- Memory

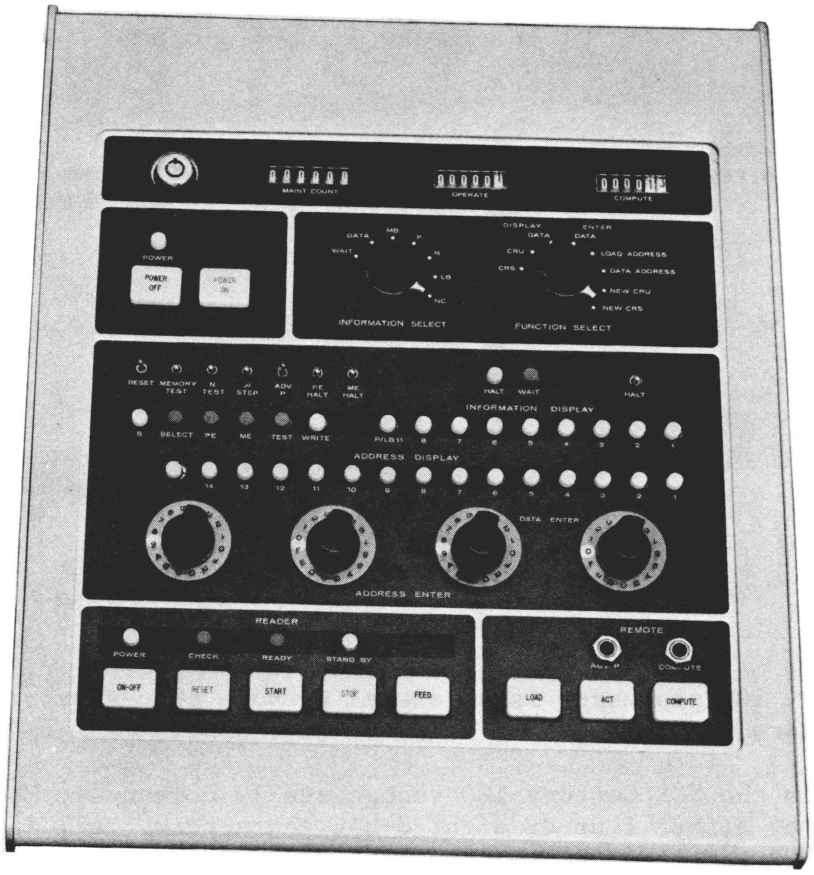
  The standard data storage capacity of the NCR Century 100 internal memory is 16,384 8-bit characters. A memory capacity of 32,768 characters is available on an optional basis. Each character in the memory has a parity bit (ninth bit) to verify the accuracy of all data accessed in the memory. The NCR Century 100 memory cycle time is 800 nanoseconds (.8 microseconds).

  The use of short, magnetic thin-film rods as storage elements gives the NCR Century 100 memory its high speed and low cost.

- Input/Output Control

  The input/output control in the NCR Century 100 processor permits 3-way simultaneity; i.e., the processor may be communicating with two peripheral units while the internal program in the processor is progressing.

- Operator's Control Panel



The operator's control panel is the functional control center for the NCR Century 100 System. This panel and its associated components occupy a dedicated position on one of the input/output trunks. The indicators, rotary switches, toggle switches, and push-button switches are logically and conveniently arranged on the panel.

The operator's control panel contains a running-time meter, a compute-time meter, and a maintenance counter. The time meters provide the user with valuable information for time studies, billing, and maintenance scheduling.

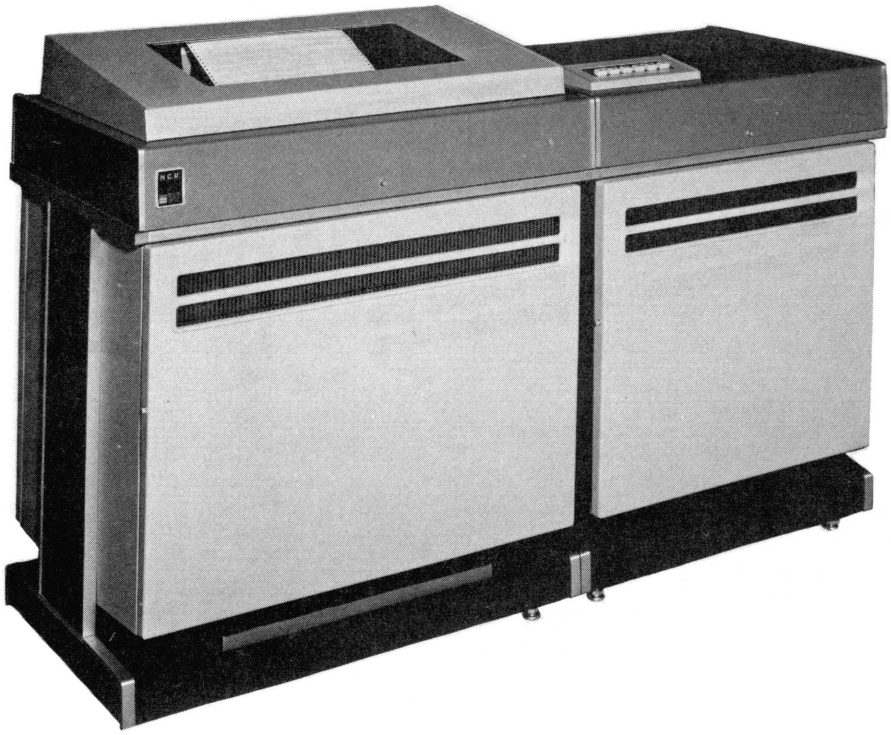
System Dual Disc

The system dual disc is a magnetic file device with random-access capabilities. It contains two discs, each with its own read/write head assembly on a movable arm. The disc packs may be independently mounted or removed.

Each disc pack has a storage capacity of 4,194,304 characters. The data transfer rate between the processor and the integrated dual disc unit is 108,000 characters per second. The average access time is 55 milliseconds; the minimum access time is 30 milliseconds; the maximum access time is 70 milliseconds. The average latency time (one-half of a disc revolution) is 21 milliseconds.

The design of the integrated dual disc unit permits easy changing of disc packs. Each disc pack is sealed to protect its recording surfaces.

System Printer



The NCR Century 100 base system has a line printer with a printing speed of up to 450 lines per minute. Each line has 132 columns (print positions). The character-set on the type cylinder of the system printer includes 26 alphabetic characters, 10 numeric characters, 28 special characters, and space.

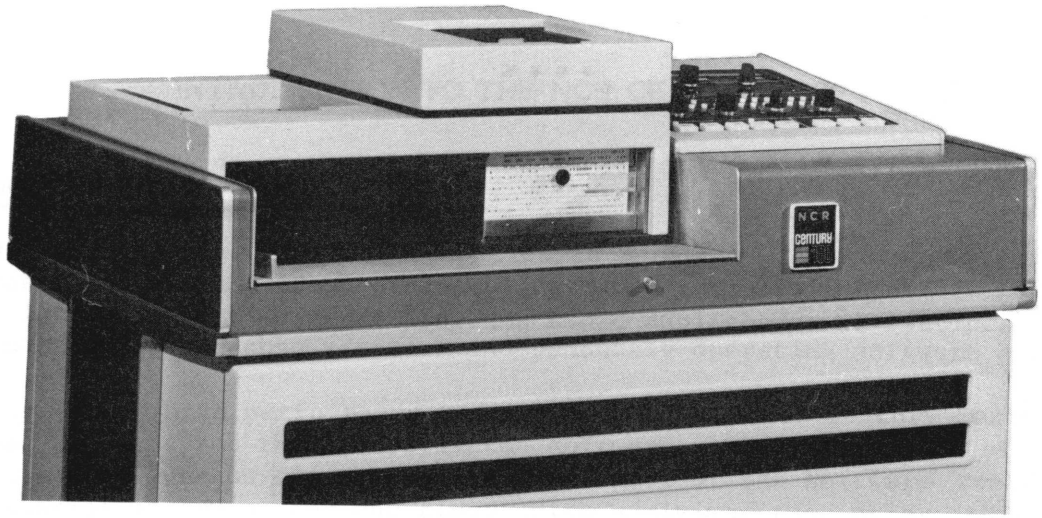
The operator can easily adjust the system printer to accept paper or forms of different widths, lengths, and weights.

An optional model of the system printer has a double set of numeric characters in each print column to permit the printing of numeric characters at a speed of up to 900 lines per minute. The type cylinder in this printer contains 26 alphabetic characters, 10 numeric characters, and 16 special characters.

System Paper Media Readers

The operator of an NCR Century 100 System uses a punched card reader or punched paper tape reader to input control information to the processor. These system paper media readers may also be used to read data files for input to users' programs.

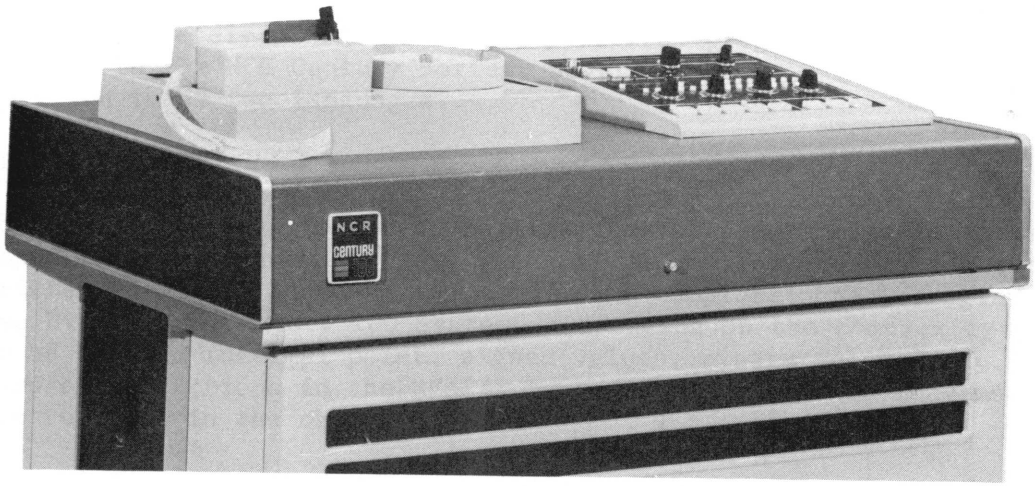
- System Punched Card Reader



The system card reader reads 300 cards per minute. The feed hopper and output stacker, which have a capacity of 1000 cards each, are easily accessible. The transport mechanism maintains constant contact and control of the cards as it guides them through the reader. This positive card control minimizes card jams and wear.

The card reader reads the cards photoelectrically, one column at a time. The hardware in the reader reduces the punch configuration from each 12-row column into an 8-bit character. (The software performs further translations, if required.)

- System Punched Paper Tape Reader



The system paper tape reader, which is basically a strip reader, accepts 5-, 7-, or 8-channel punched paper tape. However, rolls of paper tape up to 300 feet long may also be processed on this paper tape reader. The reading speed is 1000 characters per second.

* * * *

INTRODUCTION TO THE NCR CENTURY SOFTWARE

THE NCR CENTURY SOFTWARE PHILOSOPHY

The software for the NCR Century Series is a highly integrated and interdependent set of individual programs, designed to fulfill specific tasks. These interdependent software programs avoid duplication of routines and are much more efficient than a set of independently operating software programs.

The key to the high performance and low cost of the NCR Century Series lies in the use of discs and the disc-oriented software. The software is of modular design and resides on discs; only a small part of the software resides in the internal memory at any one time. With a minimum time delay, this software calls from disc into internal memory any additional software when it is needed.

Using NCR's software and discs with even the smallest member of the NCR Century Series provides a very desirable upward compatibility of programs. If a user upgrades to a more powerful system within the NCR Century Series, he may recompile his existing source programs to use them with improved efficiency.

NCR CENTURY 100 BASE SYSTEM SOFTWARE

This publication briefly explains the functions and features of the major software items for a basic NCR Century 100 System.

Programming Aids

The machine language (object program) for an electronic data processing system is completely different from a practical programming language (source program). In the NCR Century Series, several program compilers bridge the gap between these two languages.

- NEAT/3 Compiler

  The NEAT/3 language is a near-English general programming language developed for the Century Series. The NEAT/3 Compiler accepts instructions written in NEAT/3 language and compiles a complete object program for use by the NCR Century 100 System. Depending on the complexity of the data to be handled (decimal point, signed values, binary values, etc.), some simple instructions in the NEAT/3 language may result in many machine instructions in the object program.

The programmer assigns unique reference tags to logical points within his source program. He may then refer to these points at any time in the program by merely using the names of the appropriate reference tags. This feature completely eliminates the need for the programmer to concern himself with actual memory addresses.

The NEAT/3 Compiler also facilitates the execution of major (business) functions such as collating, validating, master file updating, etc. The programmer fills in clearly defined parameter and data layout sheets for these functions. The NEAT/3 Compiler accepts the information on these sheets and generates complete program segments to perform these major functions.

Many other software routines work in conjunction with the NEAT/3 Compiler and provide a programming system of extreme simplicity and efficiency. NEAT/3 also offers convenient debugging facilities. The programmer debugs and corrects all his programs at the NEAT/3 language level, eliminating the need to code-check programs at the machine language level.

- NCR Century COBOL Compiler

The NCR Century COBOL Compiler is a system of programs that translates COBOL (COmmon Business Oriented Language, U. S. Government 1965 Specifications) into an object program for use on the Century Series.

Once it has been written in COBOL, a program may be translated into object programs for different types of computers by different COBOL compilers. COBOL uses a familiar and conventional vocabulary for its source-program statements. These source statements, which can be understood by any programmer, require few documentary remarks.

The NCR Century 100 System with its standard 16,384-character memory permits the use of a basic subset of the standard COBOL.

- NCR Century FORTRAN II Compiler

The NCR Century FORTRAN II Compiler translates programs written in basic FORTRAN language into an object program.

The FORTRAN (FORmula TRANslation) language simplifies the programming of scientific problems by permitting the programmer to use familiar mathematical notations in his source program. Since FORTRAN is a universal programming language (USASI standard), it allows the many FORTRAN programs available in the EDP industry to be used on NCR Century Series.

The FORTRAN IV language is more powerful and flexible than the FORTRAN II language. While it is not designed to operate with the NCR Century 100 System, FORTRAN IV is available for use on other systems within the NCR Century Series.

Operating Software System

The operating software for the NCR Century 100 System consists of the Input/Output Executive and the Monitor systems. In addition to their individual functions as explained below, these systems and some other software routines share in the maintenance of the NCR Century software and the system log.

- Input/Output Executive

  The I/O Executive handles all input and output functions and contains the necessary routines to permit automatic simultaneity of these functions. The I/O Executive complements the hardware to a greater degree than any other software item.

  In the past, NCR's I/O Executive systems treated each peripheral unit differently. In constrast, the I/O Executive for the NCR Century Series operates all peripheral units in a uniform manner. This uniform approach results in three distinct advantages:

  - The user's program may initiate I/O instructions without regard to the degree of simultaneity of the system or the number of I/O buffers assigned by the program.
  - The NCR Century Series easily accommodates new hardware designs and different peripheral mixes.
  - The NCR Century Series offer maximum user convenience and operating speed, as well as economical utilization of memory.

  The I/O Executive employs two major concepts: captured I/O instructions and hierarchical subroutine organization.

  - Captured I/O instructions -- The simple I/O instructions in the user's program do not directly affect the hardware. Instead, they call entire software subroutines to perform the desired I/O functions.
  - Hierarchical subroutine organization -- Any program logic that is common to two or more subroutines is organized into a lower level subroutine. This concept saves memory space by avoiding unnecessary duplication.

  The programmer directs the functions of the I/O Executive through file specification data. He fills in preprinted file specification sheets for each file used in a program. The programmer's answers to the specific questions on the file specification sheets are compiled with the source program and become part of the object program. The I/O Executive uses the programmer's answers, in their compiled form, to perform the necessary input and output functions during processing. All these functions are fully explained in a separate publication on the I/O Executive System. Following are brief descriptions of some of the functions which may be of special interest to the user.

- Opening of Files

  The I/O Executive must open input and output files before the program uses them. The programmer may let the I/O Executive open a file automatically at the beginning of a program, or he may use an instruction to open a file at any point in the program.

  To open an existing magnetic source file, the I/O Executive checks the file identification label and file date to verify the validity of the source file. To open a destination file on disc, the I/O Executive finds an expired or unused area on the disc. If magnetic tape is specified as destination media, the I/O Executive checks the dates in the file identification label of the magnetic tape to determine whether or not the tape contains protected data. A destination file on magnetic media normally cannot be opened unless the media contains expired data.

- Closing of Files

  The programmer may let the I/O Executive close a file automatically at the end of a program, or he may use an instruction to close a file at any point in the program. To close a file, the I/O Executive completes all functions connected with this file. A closed file is protected. Unless the file is reopened, it is no longer available to the program.

- Error Handling

  During the running of a program, the I/O Executive takes any corrective action that may be required to assure error-free data input and output.

- Monitor

  The Monitor software provides the user with the means to run and control the NCR Century 100 System.

  - Program Loading and Linking

    The Monitor facilitates the loading of individual programs and permits the automatic linking of a predetermined series of programs. This automatic program linking provides efficient operation of the NCR Century 100 System with a minimum need for operator intervention.

  - Dating Scheme

    The NCR Century software permits the user to generate a 3-year calendar for software use. He may then run any program selectively, depending on calendar information or other information in memory. The user-defined calendar also permits the use of relative dating. For example, the programmer may specify the number of workdays a magnetic file is to be protected for backup purposes.

- Communication

  The user enters all operational information under the control of the Monitor. The operational information for system control includes such items as current dates, peripheral unit assignment to trunk positions, request for certain programs, etc.

- System Log Maintenance

  Various portions of the operating software maintain a system log on the system disc packs. The system log of the NCR Century 100 System serves to:

  - Maintain chronological entries regarding daily operations.
  - Point out normal or abnormal operating conditions.
  - Record equipment functions or malfunctions at the time of their occurrence.
  - Assist in system failure diagnosis.
  - Provide printed reports describing the above conditions.

  The operating software automatically maintains the system log for one working day. However, software is available to copy daily logs to another disc file, enabling the user to accumulate system log data over any length of time.

- Disc Management

  The operating software controls the changing of disc packs.

  When the operator replaces one of the disc packs in the NCR Century 100 System, software automatically updates the newly mounted disc pack. The NCR Century software compares the versions of the software overlays and system-oriented data on the two system disc packs and selectively copies the latest versions to the newly mounted disc pack.

  Before the operator removes the system disc containing the current system log, the software copies the log to the remaining system disc.

Utility Routines

Utility routines perform those functions which are frequently performed during the operation of most computer systems. The availability of these utility routines saves the user valuable programming time and facilitates the operation of his system.

- Sort Program Generator

  The Sort Program Generator is actually a special purpose compiler. The programmer specifies the sorting operation on special preprinted specifications sheets and inputs his entries to the Sort Program Generator. The resulting sort program is exceptionally efficient. Each sort program is tailor-made at program running time for the system configuration being used and for the data being sorted as defined on the specifications sheets. The easy and fast operation of the Sort Program Generator eliminates the need to retain many individual sort programs on discs.

- Data Utility Routines

  A number of utility routines perform the following data-oriented tasks:

  - Copying, with many options, all types of files from any type of media.
  - Verifying data files (matching of records in two files) and printing all unmatched records.
  - Preparing dummy files for program testing.

- Media Initializing Routines

  The Media Initializing routines record required labels and software on magnetic file media.

- Source Program Utility Routines

  These routines prepare source program input from punched cards, punched paper tape, or disc for use by one of the compilers (NEAT/3, COBOL, or FORTRAN). The Source Program Utility routines sort the statements in the order specified by the programmer and output errors to the printer.

  Source Program Utility routines also build control tables (control strings) that automatically run a number of individual programs in a predetermined sequence.

- Object Program Utility Routines

  The Object Program Utility routines copy object programs from disc to disc. For example, a newly compiled and debugged program can be copied to any desired disc pack for later use. These routines also maintain both an up-to-date compiler subroutine library and an operation code generator file.

Applied Programs

NCR offers its users efficient and completely debugged programs of general interest. These programs reflect the most up-to-date methods and successful practices known in business and industry.

All applied programs consist of documented sections (subroutines) which may be modified individually for optimum operating efficiency under the varying requirements of different users.

Applied programs for the NCR Century 100 System are available in the following major categories: retail, financial, industrial, commercial, and scientific.

* * * *

INTRODUCTION TO NEAT/3

* ELECTRONIC DATA PROCESSING

In summary of the prerequisite courses, data processing is a series of operations performed on data to arrive at a useful and meaningful result. Facts, when considered individually, are simply that -- isolated facts. But once they are analyzed with other information, the facts become significant. The 1,000th sale of a machine becomes meaningful only after the sale is related to the projected goal -- 1,000 sales or 10,000 sales.

Electronic data processing quickly and accurately transforms data into useful and meaningful information. But no matter how fast and how accurate a computer or any other data processing machine is, it cannot do anything without the programmer telling it what to do. To solve a problem using an EDP system, the programmer should follow these steps:

1. Define the problem.
2. Define the system of runs needed to solve the problem.
3. Flowchart the data flow (input files, magnetic storage files, printed reports) through all the runs in the system.
4. Flowchart each run in the system.
5. Write the source program for each run.
6. Compile the source program into an object program.
7. Debug (check) each program.
8. Debug (check) the system.
9. Run the system for production.

The prerequisite courses were concerned with steps 1 through 4. This course is concerned with steps 5 through 8 -- how to write the program and to get it ready to be run.

Before the actual methods needed to write a program can be learned, the WHY of these programming methods must be understood.

Programming methods are dependent on the manufacturer's hardware/software combination. The WHY of the compiler is basic to the understanding of programming.

WHY A COMPILER?

The major importance of a compiler in any computer system is to prepare a source program for the production run. A compiler is a software program that converts the program, as it is written, into a form that can be machine-executed. The source program is written in a near-English language that the compiler can understand, and the compiler translates this program into one that the processor can understand.

Obviously, the compiler must be told everything it has to know before it can translate a source program. It must be told what the data looks like, what is to be done with it, where the results are to be stored, and in what form these results are to be stored. To do this, a programmer writes a program, putting his thoughts into the expressions and the format that the compiler understands.

The program that a programmer writes is called a source program. He writes it in the language understood by the particular compiler he is using. The compiler designed to prepare a program to be run on an NCR Century computer is the NEAT/3 Compiler, and its associated language is the NEAT/3 language.

The NEAT/3 Compiler is a software program that accepts the source program as the programmer has written it. It looks at the format and the coding of each source statement within the program, determines-- as far as possible-- if the statement is correct, translates the correct statements into machine language, and flags the error statements. The translated program, the output of the compiler run, is called an object program. This object program, when free from errors, will operate on live data.

TYPES OF INSTRUCTIONS IN A SOURCE PROGRAM

A source program contains three types of instructions: compiler control instructions, data definitions, and procedural instructions.

Briefly, the control instructions -- compiler control and data definitions -- supply special information to the compiler and direct the compiler in its operation. Control instructions greatly influence the final structure of the program; however, they never become part of the object program.

On the other hand, procedural instructions make up the logic flow of the program. When translated into machine language, they become the object program.

Data Definitions

Data definitions define the data to the compiler. For instance, a data definition might contain the following information: a data field called BALANCE is eight characters long, two of which are decimal positions and one of which is a sign position. In other words, BALANCE looks like $\pm$xxxxx$_{\wedge}$xx (the caret indicates a decimal point).

Data definitions not only tell the compiler how the data will look when it is input, but they can also tell the compiler how the data is to look when it is output. For instance, a data definition may tell the compiler that BALANCE is to be edited before it is printed; that is, it is to contain a dollar sign, a decimal point, and the proper plus or minus sign. Hence, BALANCE looks like

±$xxxxx.xx each time it is printed.

Data definitions also tell the compiler when the data is introduced into the system, that is, either now, during the compilation run or later during the production run.

- Constant Data

  Data entering the system at compilation time is called a constant. This data usually is an unchanging value which the program may repeatedly need during program execution. For example, interest rates and dividend percentages can enter the system as constants. This type of data is built into the program and is always available during program execution.

- Variable Data

  Data entering the system during the production run is usually variable data. For example, John P. Depositor has $1,463.79 in his checking account, and Robert M. Spender has $29.03 in his. This type of data is stored on some media external to the program, for example, disc, magnetic tape, punched cards or paper tape, or remote terminals connected to an online system. This data is available only after the program first calls it into the system during the production run.

Procedural Instructions

Procedural instructions are another type of instruction in the source program. Procedural instructions make up the logic flow of the program. Where control instructions direct the compiler operations, procedural instructions direct processor operations. Procedural instructions tell the processor to add, subtract, move data to another location, print a record, store data on a disc, etc.

Hence, these procedural instructions are those that the compiler translates into machine language. This translation, the compiler output, is called an object program and is the program in control of a production run.

Because of the nature of the NEAT/3 Compiler, object programs differ greatly from their source programs. For example, a programmer may code a procedural instruction such as ADD A to B. During compilation, the compiler looks at the data definitions for A and B. A has two decimal positions; B has four. A has a sign; B does not. Therefore, the compiler generates the extra instructions needed to align decimal points and to add a plus sign to B. In other words, the NEAT/3 Compiler generates the instructions that the processor needs to make A and B look alike before they are added. This generated coding becomes an integral part of the object program.

Compiler Control Instructions

Compiler control instructions are another type of instruction in the source program. Compiler control instructions direct the compiler to perform special tasks. For instance, if Program A is too long to reside entirely in memory at run time, the programmer may instruct the compiler to construct the program so that parts of the program can be stored on disc while the main program is in control. Then, as each stored part is needed, it is called into memory.

This part temporarily resides in the same area that the next part will occupy when it is called into memory at a later time during processing.

The compiler control instructions do not instruct the processor; they instruct the compiler. Hence, they are not translated into machine instructions. They do not become part of the object program, but they do affect the structure of the object program.

STRUCTURE OF THE SOURCE PROGRAM

The NEAT/3 Compiler requires the programmer to ensure that the source statements in his program are presented to the compiler in a prescribed sequence. This sequence assures the compiler that it will be informed of pertinent details before it attempts to translate the procedural instructions into machine instructions.

This sequence of source statements follows:

Compiler Control Statements

These statements are introductory statements that inform the compiler of the program name and version number and of the hardware configuration upon which this program is to be run.

File and Data Description Statements

These statements completely describe the files of data to be processed and the files of data to be output.

Constants and Working-Storage Description Statements

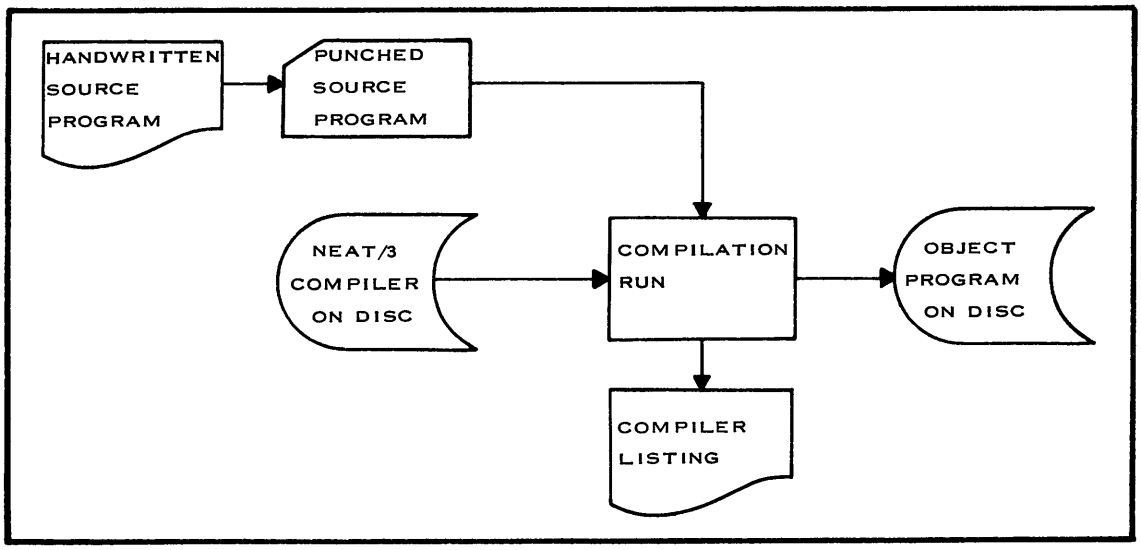
These statements inform the compiler of all the working-storage areas that the program requires during its execution.

Procedural Instruction Statements

These statements are arranged in the logical order required to manipulate the data and to produce the desired results.

STEPS TO COMPILE A PROGRAM

The following is a general and non-detailed diagram of the compilation process.



To compile a program, the programmer first writes his source program using the language and format that the compiler recognizes.

He then has this handwritten source program encoded onto punch cards, paper tape, or magnetic tape.

The encoded source program is now ready to be compiled. The Source Program Utility Routines (SPUR) first perform precompilation functions to the source program, e.g. sort the source lines into the prescribed sequence. The NEAT/3 Compiler then enters the system from disc. As mentioned before, the compiler looks at the format and the coding of each source statement within the program, determines, as far as possible, if the statement is correct, translates the correct statements into machine language, and flags the statements in error.

The compiler then outputs the object program onto a disc and prints a listing which contains a copy of the source program, a list of the errors it has found, and a memory map of the object program.

The programmer must now debug the program (or correct any errors in it that may hamper program execution). However, this subject of debugging is discussed later. For the present, the actual coding of a source program is the primary concern.

* * * *

DATA CONCEPTS

INTRODUCTION

A programmer is to write a program that will access prescribed data, process that data, and output the desired results. To do this, he must describe the format of both the data to be processed and the desired results. These descriptions tell the compiler exactly which instructions it must generate to make the procedural instructions complete.

Because of the nature of the NEAT/3 Compiler, the programmer can write his program in a language that is similar to English. This English-oriented language is simple to code, for it is divorced from actual hardware instructions. The NEAT/3 Compiler assumes this burden of building the precise hardware instructions. For the compiler to do this, the programmer must inform it of all the details of the data to be processed, of the desired output, and of the system configuration on which the program is to be run.

FILES

The concept of a file is one of the most basic concepts in data processing. All input data and output data enters and exits the system through files. This concept of a file may be likened to that of a file cabinet in an office. In this cabinet are systematically arranged records into which a secretary may enter information or from which she may obtain information. For instance, Peter S. James called his boss this morning and reported that he would not come to work today because his wife just gave birth to a baby. The secretary in the personnel office locates Peter S. James' record in the file and records his absence. She also updates the record to show an increase in the number of Peter S. James' dependents.

The concept of an electronic data processing file is quite similar to the above example. A file is a group of records that pertain to a common subject and that are written on the same media. This media may be punched cards, paper tape, magnetic tape, disc, or any other storage media.

Types of Files

Consider two types of files -- master files and transaction files. Those files that contain permanent data are termed master files. For instance, each record in an employee's master file may contain the employee's name, employee number, social security number, weekly salary, salary-to-date, income-tax-to-date and other pertinent information. The data in these master records can be changed (or updated) periodically.

The data used to update these master records is contained on transaction records. All the transaction records that are needed to update the master file are collectively termed the transaction file.

Consider a typical update run. Peter J. Swanson goes to his bank and deposits money into his checking account. The bank punches a record of this transaction into paper tape. This information and all the other transactions of the day -- the transaction file -- are input to an update run. The update program reads a transaction record and gets the corresponding master record. This pairing is accomplished by comparing the keys (e.g., account number, employee number, or stock number) in both the transaction and master files. When the appropriate master record is found, the program updates it by using the information in the transaction record. The new master record now becomes part of the updated master file.

Since master files are permanent and since they may be updated frequently, they are usually stored on magnetic media rather than on paper media. Transaction files, however, are generally used only once; e.g., after a bank posts a deposit to an account, the transaction record is no longer needed. These files may be kept for a few days as backup and then easily destroyed.

The details of file structure and definition are covered in the file concept sections of this manual. However, this brief discussion has been necessary to fully understand the following discussion of data concepts.

File Storage Codes

When data is recorded, it is encoded into the language of the machine that recorded it. Thus, a punched card code (usually Hollerith code) differs from a punched paper tape code (of which many codes are available).

The NCR Century Series always expects the data in memory to be coded in USASI code. This means that any data not recorded in this code must be translated before it is presented to the program in memory.

To illustrate this need for a translation, suppose that a transaction file is input from punched cards and is recorded in the Hollerith code. If this file were to be input directly into memory, the computer would use the Hollerith-coded data as if it were USASI-coded data. Hence, if an input record contained the name JO, the computer would interpret it as being the letters AF.

Obviously, the data must be translated. The NEAT/3 language provides a simple way to do this. When the file is defined to the compiler, the programmer specifies on which media and in which code his data is to be input. He also specifies on which media and in which code his updated data is to be output. The compiler does the rest. It includes in the object program a translation table for each code the programmer has specified and a translation routine that uses these tables. During the production run of the object program, software uses these translation tables to ensure that the input data is translated into the USASI code and that the output data is translated into the storage code the programmer has specified.

This translation routine is one of the many ways that the NEAT/3 language and the NEAT/3 Compiler aid programmers. The following discussion takes a deeper

look at this language and shows how the compiler incorporates the data descriptions into an object program, thereby easing the coding effort of the programmer.

* * * *

DATA DESCRIPTIONS

Introduction

The NEAT/3 Compiler requires that data used by the main program be described before any of the procedural instructions in the program are presented. Data used only in an overlay must be described before any of the procedural statements in that overlay are presented. This description is then available to the compiler so that it can construct the correct machine instructions from the near-English procedural instructions written by the programmer.

For instance, the compiler relieves the programmer of the burden of keeping track of the actual memory address of each unit of data in the program. Through data definitions, the programmer informs the compiler of the names he has assigned to those units of data that the object program is to manipulate. This permits the programmer to write his procedural instructions using the names of the data instead of memory addresses; i.e., the programmer writes ADD DEPOSIT to BALANCE instead of ADD contents of memory address 0092 to contents of memory address 3751.

The following discussion is of the effect data description statements have on the object program and on the actual programming of this object program. This discussion considers two kinds of data descriptions. One kind is the description of data that is external to the program, i.e., data either input to or output from the program. This data is termed a record since the data usually is stored as a permanent record within a file.

The other kind of data description that this discussion considers is the constant or fixed data that may be repeatedly needed during program execution. This data is defined as being in an area since the data is built into the object program and takes up an area in memory during program execution.

Both records and areas can be subdivided into smaller units of data called fields. Therefore, data can be defined as being either a record, an area, or a field.

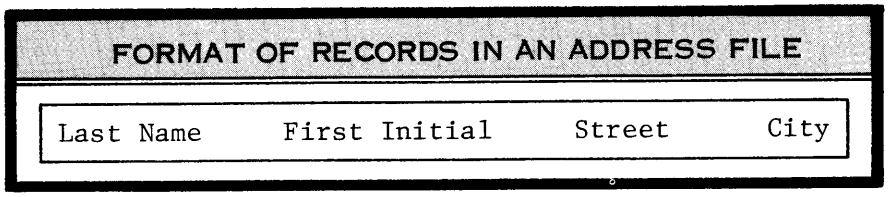
Records

Data can be defined as being a record. A record is an organized group of significant facts about a particular subject. For instance, a company may have a record of each employee which contains the employee's name and address. The company gathers the records of all the employees and places them on a common media -- punched paper tape, punched cards, disc, or magnetic tape. This group of records is then called a file. A file is a group of records in like-format written on the same media.

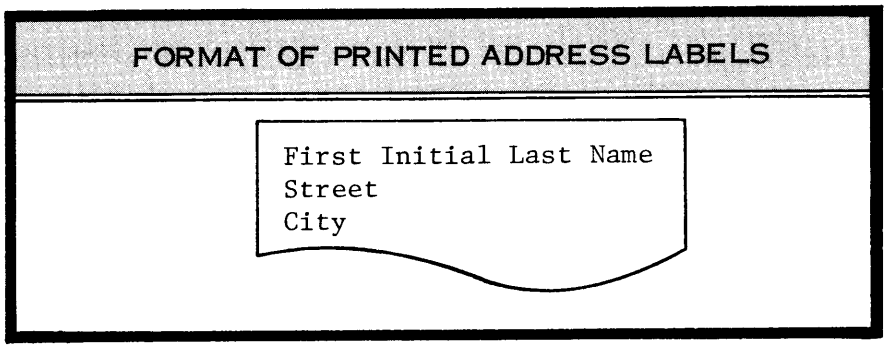
- Example of the Use of Records in a Program

A simple but typical program might get records from a name and address file, reformat these records, and print address labels. Let's consider this program.

The records in the input file have the following format:

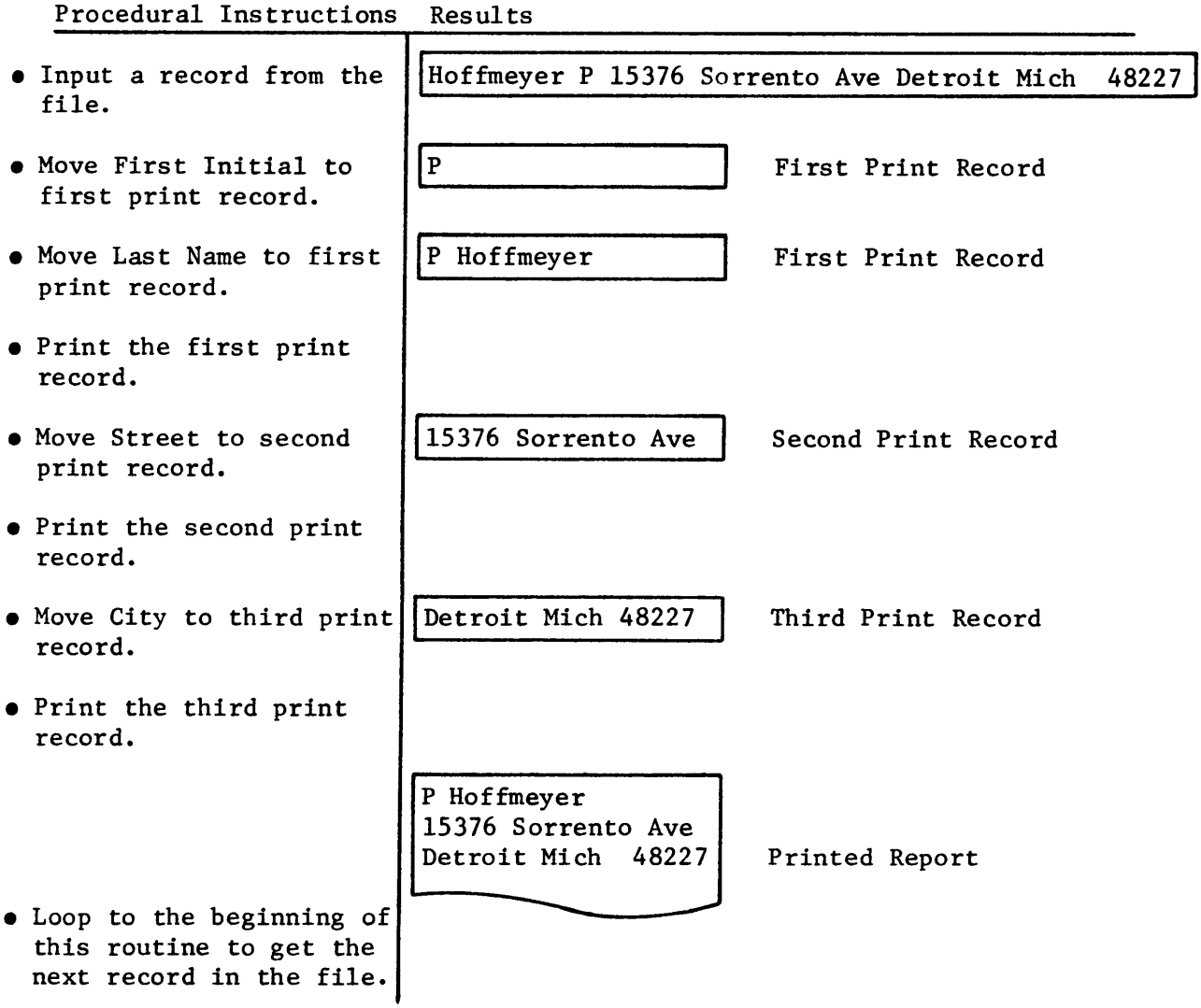


From the information recorded in this file, the program is to construct address labels printed in the following format.



Therefore, the programmer must construct a print file into which data to be output is to be stored. The format of the records in this print file must reflect the format of the address label. Since three lines of information are needed to complete one address label, the print file requires three different record formats -- the first for the person's name, the second for his street address, and the last for his city address. Thus, each record in this print file will be output as one printed line of data.

The following steps outline the procedural instructions needed to obtain the printed address labels.

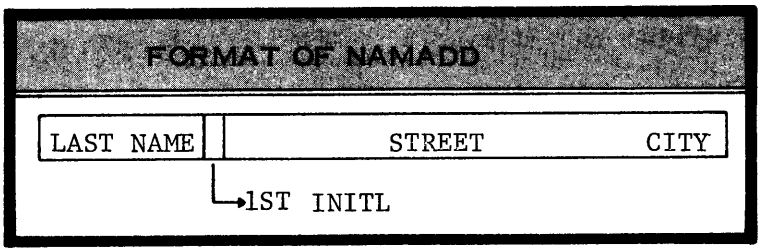


The programmer can easily see what the above procedural instructions have done, for he can look at or mentally picture both the arrangement of the data in the name and address records and also the desired arrangement of the data in the print records. However, since the compiler cannot see or interpret these pictures, the procedural instructions are worthless without a description of the data. The programmer must convey to the compiler -- by words, pictures, or codes -- the characteristics he sees and knows about the data.

The programmer describes this data to the compiler in the predefined language and format that the compiler expects. However, for now, full English sentences are used to describe those characteristics of data that the compiler must know to understand the procedural instructions. The data is described as follows.

- Input File

The input file consists of records referenced by NAMADD. Each of these records are 51 characters long.



The first 10 characters of NAMADD are collectively called LASTNAME.

The eleventh character of NAMADD is called 1STINITL.

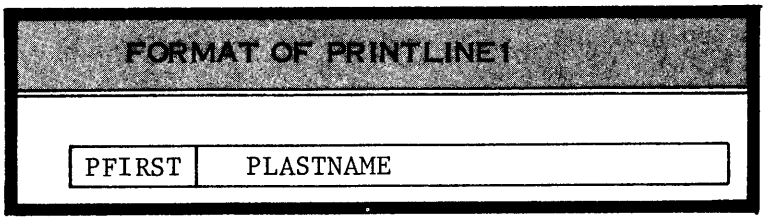
The next 20 characters are called STREET.

The last 20 characters are called CITY.

- Output File

The output file consists of three record formats.

The first record format is 12 characters long and is referenced by PRINTLINE1.



The first 2 characters of PRINTLINE1 are called PFIRST.

The next 10 characters of PRINTLINE1 are called PLASTNAME.

The second record format is 20 characters long and is referenced by PRINTLINE2.

The third record format is 20 characters long and is referenced by PRINTLINE3.

- Procedural Instructions

After the description of the input file and the output file are given to the compiler, the procedural instructions are made more meaningful.

- Input NAMADD
- MOVE 1STINITL to PFIRST
- MOVE LASTNAME to PLASTNAME
- Print PRINTLINE1
- MOVE STREET to PRINTLINE2
- Print PRINTLINE2
- MOVE CITY to PRINTLINE3
- Print PRINTLINE3
- Loop to the beginning of this routine and read the next NAMADD

The compiler can now consult the data descriptions and make sense out of the procedural instructions. Since it is the compiler's job to organize the object program, it knows where in memory the data will be located during program execution. It, therefore, takes our simple procedural instruction (MOVE LASTNAME to PLASTNAME), changes each data reference to a memory address of where that data can be found, and states these instructions in a machine format that means the following: starting at location xxxx, pick up 10 characters and move these characters to location yyyy.

It is easy to see exactly why data descriptions of records are so necessary to an object program. The very nature of the NEAT/3 Compiler allows a programmer to write simple procedural instructions -- instructions that, through data descriptions, the compiler interprets and changes into a correct machine language.

Areas

The previous discussion was of records, that is, data which is external to the object program and which is called into memory during program execution. However, some data is built into the object program. This place in the object program is termed an area, for it contains data which occupies an area in memory during program execution. Areas reserved in memory serve two purposes.

- Working Storage Areas

  Area definitions can be used to reserve space in memory for working storage areas. Values may be temporarily stored within these areas during program execution. For instance, the results of a mathematical calculation may be stored in an area to be used later in the program.

- Constants Areas

  Areas may also be used to contain constants. Data entering the system at compilation time is called a constant. This data usually is an unchanging value which the program may repeatedly need during program execution. For instance, a programmer may define an area to contain page headers (constants). During program execution, these constants can be moved to a printline area each time a new page is to be printed. This type of data, an area, is built into the program and is always available during program execution.

  Since a program will probably require more than one constant, the programmer may instruct the NEAT/3 Compiler to reserve a large area which is to contain a logical group of constants. He may then break this large area into many smaller areas (or fields) into which he will tell the compiler to store these constants.

  Each program may contain as many distinct areas as are needed.

- Example of a Program Using Both Types of Areas

  For example, suppose a programmer is to write a program that is to read account records, calculate the interest owed, and accumulate the total interest owed on all accounts. Before he writes the procedural instructions for this program, he must first define the data.

- Input File

The input file is made up of records called ACCREC.

FORMAT OF ACCREC

	BALANCE
xxxxxxxxxxxxxxxxxxxx	xxxxxxxx^xx

Each ACCREC is 30 characters long.

The last 10 characters of ACCREC contain the account's balance and are referenced by BALANCE.

BALANCE contains a decimal number that has two decimal positions.

- Reserved Area

The reserved area is 21 characters long and is referenced by RESERVAREA.

FORMAT OF RESERVAREA

RATE	TOTAL	WRKSTORAGE
^035	00000000^00	000000^00

The first field within RESERVAREA is referenced by RATE. It is three characters long and contains .035, a constant value. (Decimal points, unless they are in an editing mask, do not occupy a character position in memory and, therefore, are not reflected in this 3-character length.)

The next 10 characters within RESERVAREA are referenced by TOTAL. TOTAL has two decimal positions. Initially, this field contains zeros.

The last field within RESERVAREA is referenced by WRKSTORAGE. It is eight characters long and has two decimal positions. Initially, this field contains zeros.

- Procedural Instructions

  Using the data definitions of the input file and the reserved area, a programmer can now write the procedural instructions needed to manipulate his data.

  - Input ACCREC
  - MULT BALANCE times RATE, store results in WRKSTORAGE
  - ADD WRKSTORAGE to TOTAL
  - Loop to beginning of this routine and read the next ACCREC

  The compiler can now read the procedural instructions, relate them to the description of the data, and produce an object program that would output the desired results.

Fields

As previously defined, the data that is external to the object program is stored in records; the data that is built into the object program is stored in areas. Records and areas can be further subdivided into fields. A field is a unit of data within a record or an area. The previous examples of areas and records have shown fields. Consider these examples again.

- Example of Fields Within an Area

  A 21-character area referenced by RESERVAREA was defined as having three fields -- RATE, TOTAL, and WRKSTORAGE. The format of this area follows:

FORMAT OF RESERVAREA

RATE	TOTAL	WRKSTORAGE
^035	00000000^00	000000^00

Each of these fields - RATE, TOTAL, and WRKSTORAGE -- is a unit of data that can be individually accessed during processing.

- Example of Fields Within a Record

The following record contains many fields. This record is a daily sales record from each store in a grocery chain. The fields in this record are Store Code, Manager, Date, Groceries, etc. The format of this record follows:

DAILY SALES RECORD FROM ONE STORE IN A GROCERY CHAIN

Store Code	Manager	Date			Groceries	Produce	Meat	Dairy	Misc.	Daily Total
		Da	Mo	Yr						
274	Peterson	22	09	68	3196.43	572.86	1205.69	529.73	574.93	6079.64

- Example of Fields Within Another Field

A field may be part of another field, or it may include other fields within its own definition. For instance, the field Date in the Daily Sales Record is divided into three separate fields -- Day, Month, and Year.

Date		
Da	Mo	Yr
22	09	68

Each field that is to be individually accessed during processing must be assigned a name. If Date is to be treated as a 6-character unit of information, it need not be further divided into Day, Month, and Year. Only one name -- Date -- need be assigned to the field.

Date
xxxxxx

If the fields within Date are to be referenced individually, then these fields must be assigned a name, i.e., Day, Month, and Year.

Date		
Da	Mo	Yr
xx	xx	xx

If only Day and Month are to be referenced individually, Year need not be specifically defined. Nevertheless, since the Day, Month, and Year fields are contained in each record, the Date field must be defined as containing six characters.

Date		
Da	Mo	
xx	xx	xx

DATA TYPES

As the previous examples have shown, the type of data used -- alphabetic or numeric -- varies for different programs, for different records or areas, and even for different fields within the same record or area. The two basic types of data that can be processed are alphabetic and numeric. However, the NEAT/3 Compiler allows the programmer to have numeric data stored in his option of various numeric codes.

Some types of numeric data require the compiler to generate more software to complete the procedural instructions than do other types; i.e., signed numeric data requires more software to arithmetically manipulate it than does unsigned numeric data.

Some types of numeric data require less storage space than do other types. For instance, one type of numeric data allows the programmer to conserve storage space by storing data fields in a condensed (packed) form.

The programmer must inform the compiler of the exact type of data that is to be processed, whether it be alphanumeric or one of the various numeric types. Following is a list of the allowed types of data:

DATA TYPES RECOGNIZED BY THE NEAT/3 COMPILER

USASI Characters	For alphanumeric data
Unsigned Decimal Signed Decimal Unsigned Packed Decimal Signed Packed Decimal Binary Hexadecimal	For numeric data
Editing Mask	For data to be printed

* * * *

The following illustration shows the bit representation of each character in the alphanumeric and numeric types of data:

NCR CENTURY CODE CHART

B_8–B_5 ↓ / B_4–B_1 →	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0001	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0010	2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0011	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0110	6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	7	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	DEL

DATA TYPE	ALLOWABLE CHARACTERS	BIT REPRESENTATION
USASI CHARACTERS	ALL CHARACTERS SHOWN IN CHART	8 BITS
UNSIGNED DECIMAL	0–9	8 BITS
SIGNED DECIMAL	0–9, +, AND –	8 BITS
UNSIGNED PACKED	0–9	4 BITS
SIGNED PACKED	0–9, +, AND –	4 BITS

Alphanumeric Characters

Data can be represented as alphanumeric characters. This type of data can contain any combination of the letters A to Z, the numerals 0 to 9, and certain special characters. Rows 2 through 5 in the NCR Century Code Chart show the coded representation of each of these letters, numerals, and special characters.

Names, addresses, report headers, account numbers, etc. can all be defined as being alphanumeric characters. These characters cannot be used in mathematical operations. However, they can be moved, compared, etc. during program execution.

Unsigned Decimal Data

Data can be represented as an unsigned decimal number. An unsigned decimal number is a data string that contains only the numeric characters 0-9 and that has no + or - sign associated with it. Each unsigned decimal character is recorded in eight bits whether it be recorded in memory or on a storage media. The data may be an integer or a decimal value.

All unsigned decimal data is considered by the compiler to be a positive decimal number. When the compiler encounters a procedural instruction that is to manipulate this type of data, it generates in the object program the coding needed to handle positive numbers only. Therefore, the programmer should define as being unsigned decimal characters only those data strings that he knows will always remain positive during processing and, therefore, will never require a sign.

Since the compiler does not have to generate the instructions needed to handle a negative field, the software required to manipulate unsigned decimal data is less than that required to handle signed decimal data. However, the generated logic for manipulating signed fields with other signed fields is shorter than the logic for manipulating signed fields with unsigned fields. In other words, it is better to manipulate two like fields rather than two unlike fields.

For example, a programmer is to write a program that requires the number .035 to be used many times during processing. He wishes this number to be incorporated into the object program as being an unsigned decimal number. Therefore, when he defines his data to the compiler, he incorporates this number into the area reserved for constants. To do this, he specifies that the constants area contains a three-character field referenced by RATE. The data contained in RATE is 035, an unsigned decimal number that has three decimal positions. The compiler will then build 035 into the object program, allowing eight bits for each numeric character. It also generates the extra coding to complete the procedural instructions that manipulate this data. (For example, it generates the extra coding needed to align the decimal point in RATE with the decimal point in the data manipulated with RATE.)

In the previous example, the unsigned decimal number was used as a constant. A programmer may also define an unsigned decimal number as a field within a record. For instance, a programmer is to write a program that updates the master records for customer savings accounts. His transaction records must add the deposit or subtract the withdrawal from the balance left in the customer's account record. Since the deposit or withdrawal always reflects a positive amount, the programmer defines this field as containing an unsigned decimal number. To do this, he defines to the compiler that a field referenced by AMOUNT will be contained in each transaction record. The data in AMOUNT has two decimal positions and is an unsigned decimal number. The compiler accepts this format; with this information, it generates the extra coding needed to complete the procedural instructions that manipulate this data. Then during processing, a transaction record and its corresponding master record are accessed, and the master record is updated with the information contained in AMOUNT.

Signed Decimal Data

Data can be represented as a signed decimal number. A signed decimal number is a data string that contains the numeric characters 0 to 9 and that has a + or - sign associated with it; for instance, +32.9 is a signed decimal number. Except for the presence of a sign, signed decimal data has the same characteristics as does unsigned decimal data.

When the compiler encounters a procedural instruction that is to manipulate this type of data, it generates in the object program the coding needed to make the procedural instructions complete. This coding will handle not only positive numbers but also negative numbers. Also, if the signed decimal number is to be manipulated with an unsigned decimal number, the compiler generates the extra coding needed to make both data fields alike, that is, to add a sign to the unsigned decimal number. Because of this special software, a programmer should define as signed decimal numbers only that data whose nature demands an associated sign. If the data does not demand an associated sign, he should define the data as an unsigned decimal number.

Unsigned Packed Decimal Data

Data can be represented as an unsigned packed decimal number. An unsigned packed decimal number is a data string that contains the numeric characters 0-9 and that has no sign associated with it.

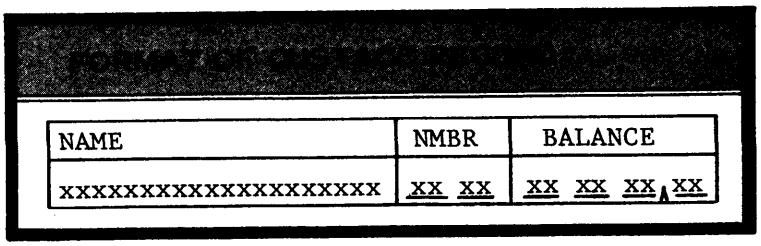
Two unsigned packed decimal characters are recorded in eight bits whether they be recorded in memory or on an external storage device.

Data in an unsigned packed decimal field must be moved to an unsigned, unpacked field before mathematical operations can be performed on the data. Generally, unsigned packed data is used to conserve space on an external storage device.

- Example of a Program Using Unsigned Packed Decimal Data

For example, a programmer is to write a program that will update customer accounts. Because he has a large master file, he decides to conserve storage space on the disc. The technique he chooses to conserve this space is the use of packed data. He defines this data in the following manner:

- Input 1: Master File



NAME	NMBR	BALANCE
xxxxxxxxxxxxxxxxxxxx	xx xx	xx xx xx$_{\wedge}$xx

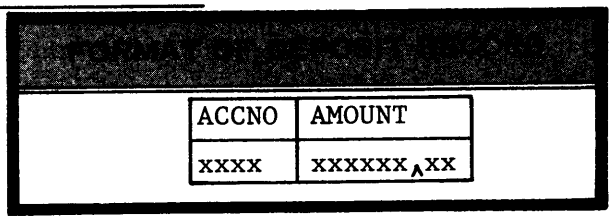
The current master file is made up of the records referenced by CUSTACC. Each record contains twenty-six 8-bit characters.

The first 20 characters are referenced by NAME.

The next two characters are referenced by NMBR. This field contains four unsigned packed decimal characters.

The last four characters are referenced by BALANCE. This field contains eight unsigned packed decimal characters, two of which occupy decimal positions.

- Input 2: Transaction File



ACCNO	AMOUNT
xxxx	xxxxxx$_{\wedge}$xx

The transaction file is made up of records referenced by DEPOSIT. Each record contains twelve 8-bit characters.

The first field of DEPOSIT is four characters long and is referenced by ACCNO. This field contains unsigned decimal characters.

The second field of DEPOSIT is eight characters long and is referenced by AMOUNT. This field also contains unsigned decimal characters, two of which occupy decimal positions.

- Output: New Master File

FORMAT OF NEWCUSTACC RECORD

NAME	NMBR	BALANCE
xxxxxxxxxxxxxxxxxxxx	xx xx	xx xx xx^xx

From the information contained in the transaction records, the programmer is to update the current master file and to create a new master file. Each record in this new master file is 26 characters long and is referenced by NEWCUSTACC.

The first field in NEWCUSTACC is 20 characters long and is referenced by NAME.

The second field in NEWCUSTACC is two characters long and is referenced by NMBR. This field is to contain four unsigned packed decimal characters.

The third field in NEWCUSTACC is four characters long and is referenced by BALANCE. This field is to contain eight unsigned packed decimal characters, two of which occupy decimal positions.

- Reserved Area

FORMAT OF WORKAREA

WRKAREA1	WRKAREA2
xxxxxx^xx	xxxxxx^xx

The programmer tells the compiler to reserve a memory area 16 characters long referenced by WORKAREA.

The first field of WORKAREA is eight characters long and is referenced by WRKAREA1. This field is to contain unsigned decimal characters during program execution.

The second field of WORKAREA is eight characters long and is referenced by WRKAREA2. This field also is to contain unsigned decimal characters during program execution.

- Procedural Instructions

Remember that on the NCR Century 100 no arithmetic operations can be performed on unsigned packed numbers. Therefore, the procedural instructions must move the data from an unsigned packed field into an unpacked field before the data is arithmetically manipulated. When the compiler encounters this MOVE, it generates in the object program the extra instructions needed to convert the packed data to unpacked data. The programmer writes his procedural instructions as follows:

Procedural Instructions	Action
Input CUSTACC.	JOHNSTON JAMES R | 59 27 | 14 09 01$_{\wedge}$70 — CUSTACC
Input DEPOSIT.	5927 | 002816$_{\wedge}$40 — DEPOSIT
MOVE BALANCE to WRKAREA1.	140901$_{\wedge}$70 | — WORKAREA
MOVE AMOUNT to WRKAREA2.	140901$_{\wedge}$70 \| 002816$_{\wedge}$40 — WORKAREA
ADD WRKAREA2 to WRKAREA1 and store result in WRKAREA1.	143718$_{\wedge}$10 \| 002816$_{\wedge}$40 — WORKAREA
MOVE WRKAREA1 to NEWCUSTACC's BALANCE.	xxxxxxxxxxxxxxxxxxxx | xx xx | 14 37 18$_{\wedge}$10 — NEWCUSTACC
MOVE CUSTACC's NAME to NEWCUSTACC's NAME.	JOHNSTON JAMES R | xx xx | 14 37 18$_{\wedge}$10 — NEWCUSTACC
MOVE CUSTACC's NMBR to NEWCUSTACC's NMBR.	JOHNSTON JAMES R | 59 27 | 14 37 18$_{\wedge}$10 — NEWCUSTACC
Output NEWCUSTACC.	
Loop to beginning of this routine to read another CUSTACC and DEPOSIT.	

Briefly then, unsigned packed decimal characters conserve space on external storage devices. However, when this data is brought into memory, it must be converted to an 8-bit decimal code by a MOVE instruction before it can be arithmetically manipulated or printed.

Signed Packed Decimal Data

Data can be represented as a signed packed decimal number. A signed packed decimal number is a data string that contains the numeric characters 0-9 and that has a + or a - sign associated with it. Except for the presence of a sign, signed packed decimal data has the same characteristics as does unsigned packed decimal data.

Data in a signed packed decimal field must be moved to a signed unpacked field before mathematical operations can be performed on the data. Generally, signed packed data is used to conserve space on an external storage device.

The sample program used to illustrate unsigned packed decimal characters could also be used to illustrate signed packed decimal characters. Of course, the signed packed decimal data would include a plus or a minus sign, but all other characteristics and procedures would remain the same.

Binary Data

Data can be represented as a binary number. Binary numbers use all eight bits of each memory position reserved for the value. Within an 8-bit field, the binary equivalent of the positive integer 255 can be represented. NEAT/3 language allows programmers to manipulate the binary equivalent of any positive integer ranging from 0-999,999.

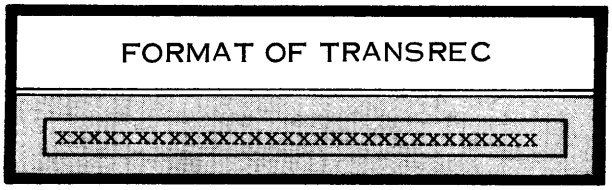
Data in a binary field can be added to or subtracted from data in another binary field, but it cannot be manipulated with data in a decimal field.

Since the use of binary numbers saves both external storage space (e.g., disc) and internal memory space, programmers can advantageously use binary data. For instance, a programmer may define a binary field into which he counts the number of iterations in an iteration loop.

- Example of a Program Using Binary Data

  Consider a complete program. Suppose a programmer is to write a program that reads records from a transaction file, assigns each record an item number, and then outputs these numbered records to a destination file. He is to assign the number 100 to the first transaction record, 101 to the second, 102 to the next, etc. His data definitions will tell the compiler the following information:

  - Input: Transaction File



The input file is made up of records referenced by TRANSREC. Each record contains 30 characters, all of which are unsigned decimal characters.

- Output: Destination File

FORMAT OF OUTTRANS

ITEM	DETAIL
xx	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

The output file is made up of records referenced by OUTTRANS. Each record contains 32 characters.

The first field is referenced by ITEM. This field is two characters in length and is to contain a binary number.

The second field is referenced by DETAIL. This field is to contain 30 unsigned decimal characters.

- Reserved Area

FORMAT OF AREA1

ITEMNMBR	AUGMENT
xx	x

The programmer tells the compiler to reserve a memory area three characters long.

The first field of this area is two characters long and is referenced by ITEMNMBR. This field is initially to contain the binary equivalent of 100.

The second field of this area is one character long and is referenced by AUGMENT. This field is to contain the binary equivalent of 1.

- Procedural Instructions

After defining the data, the programmer may now write the procedural instructions for his program.

Procedural Instructions	Action
• Input TRANSREC.	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx TRANSREC
• MOVE TRANSREC to DETAIL.	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx OUTTRANS
• MOVE ITEMNMBR to ITEM.	100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx OUTTRANS
• ADD AUGMENT to ITEMNMBR and store result in ITEMNMBR.	101 1
• Output OUTTRANS.	100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx OUTTRANS
• Loop to beginning of this routine and input another TRANSREC.	

Thus, by using binary numbers, the programmer can efficiently assign each transaction record a sequential item number. Binary numbers may also prove useful in many other user applications.

Hexadecimal Data

Data can be represented as a hexadecimal number. The hexadecimal equivalent of any positive integer ranging from 0 to 15 can be stored in four bits of memory.

Hexadecimal numbers are primarily used to define console messages which are to be displayed during program execution.

The hexadecimal numbers have a base of 16. The following chart shows the 16 hexadecimal characters, their corresponding decimal equivalents, and their bit configurations.

HEXADECIMAL EQUIVALENTS

Hexadecimal	Decimal	Bits
0	= 0	= 0000
1	= 1	= 0001
2	= 2	= 0010
3	= 3	= 0011
4	= 4	= 0100
5	= 5	= 0101
6	= 6	= 0110
7	= 7	= 0111
8	= 8	= 1000
9	= 9	= 1001
A	= 10	= 1010
B	= 11	= 1011
C	= 12	= 1100
D	= 13	= 1101
E	= 14	= 1110
F	= 15	= 1111

Note that the hexadecimal characters A to F are symbolic references and do not refer to the USASI representation of A to F. For instance, a USASI letter A is represented in eight bits as 01000001, and a hexadecimal letter A is represented in four bits as 1010.

Editing Mask

The data types just discussed concern data that is stored in memory and data that is input from and output to a storage media. However, the programmer may also define a field which is to be edited. He places over this field an editing mask which specifies the format of the data to be printed. Through an editing mask, a programmer can print currency symbols, decimal points, check-protect symbols, etc.; he can suppress from the printed page the leading zeros; he can insert + and - signs or CR and DB notations. This editing process makes the printed data more readable.

Through data definitions, the programmer defines the format of the data to be printed by using editing symbols. He defines this editing mask over a destination field in a print record or area. Then in his procedural instructions, he can MOVE into the destination field the data to be output. The software generated by the compiler then edits the data--inserting currency symbols, decimal points, and commas; inserting check protect symbols, + or - signs, CR or DB notations, etc. The edited data remains in the field until it is moved.

- Example of a Program Using an Editing Mask

  Consider this simple program which illustrates one way that the use of an editing mask can help programmers. A programmer is to write a program that prints the balance in each customer's account. Each printed balance is to contain a dollar sign, decimal point, and any applicable commas. The dollar sign is to be printed adjacent to the leftmost digit in the balance. The data definitions to the compiler tell the following information about the data:

  - Input: Master File

FORMAT OF CUSTREC

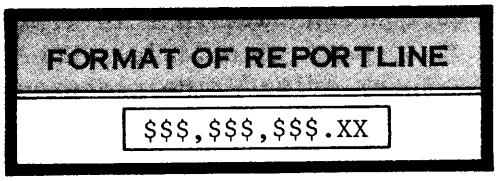
NAMADD	BALANCE
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxx$_{\wedge}$xx

The input file is made up of records referenced by CUSTREC. Each record contains 50 characters.

The first 40 characters contain the customer's name and address. This field is referenced by NAMADD and contains alphanumeric characters.

The next field contains 10 characters and is referenced by BALANCE. This field contains unsigned decimal data that has two decimal positions.

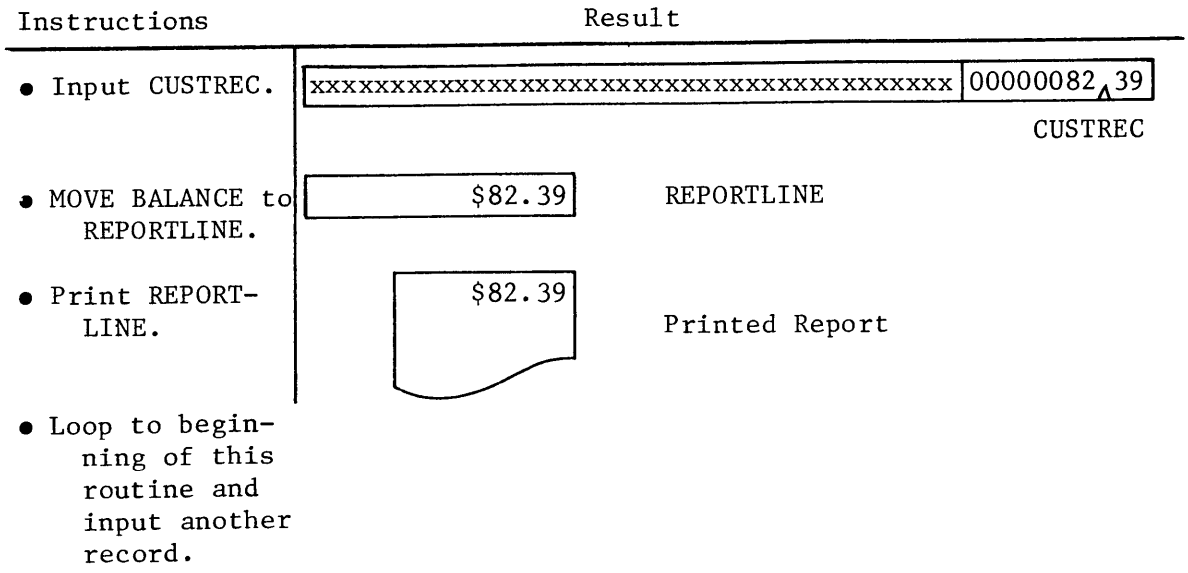
- Output: Printed Report



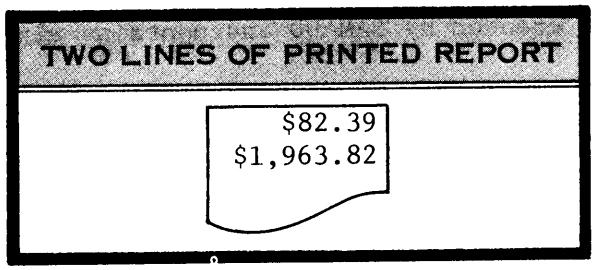
The output file contains records referenced by REPORTLINE. Each record contains 14 characters. The format of the output is shown by the editing mask defined over this 1-field record. This mask is interpreted as follows: significant digits of BALANCE will be printed; leading zeros will be suppressed; a dollar sign will be printed adjacent to the leftmost significant digit; commas and decimal point will be inserted. (The editing mask is more fully discussed later.)

- Procedural Instructions

The programmer can now write his procedural instructions:



If the next CUSTREC had a balance of 196382, this data would be edited and printed as follows:



* * * *

PROGRAMMING WORKSHEETS

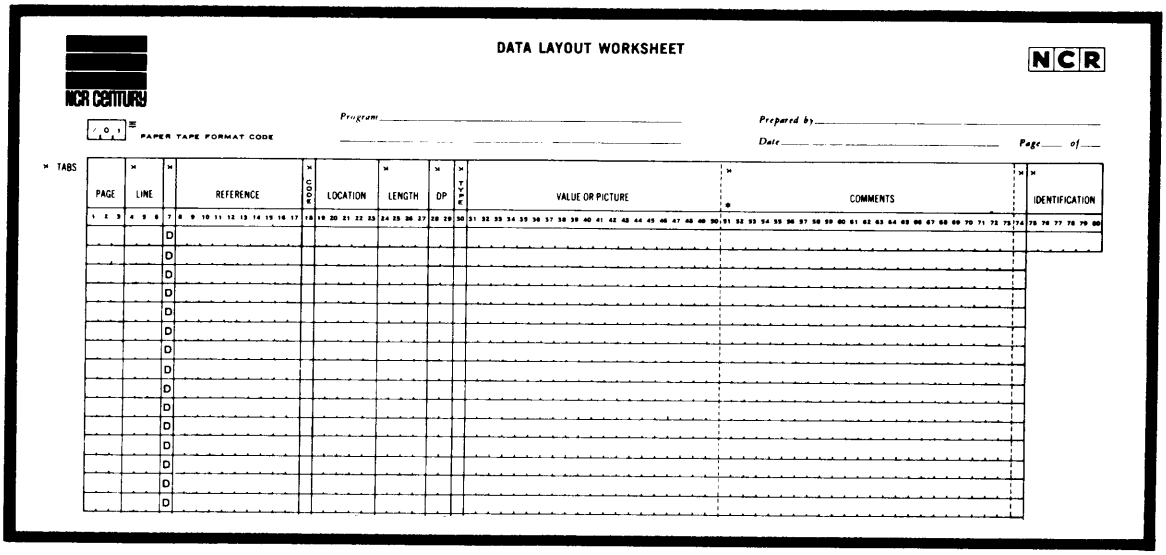
TYPES OF PROGRAMMING WORKSHEETS

The compiler needs a set of instructions to tell it what to do, what options to include, and where to put the final output. These instructions are written on programming worksheets which are divided into six general categories of program information. When the information entered on these worksheets is put on punched media, position 7 of each source line contains a 1-character code that indicates from which worksheet the source statement originated. These six worksheets and their respective codes are:

C Coding	M Major Functions
D Data Layout	P Compiler Control
F File Specification	T Table Specification

Coding and Data Layout Worksheets

The programmer writes his source-program instructions on coding and data layout worksheets. An illustration of this type of worksheet follows:


DATA LAYOUT WORKSHEET

NCR CENTURY

NCR

PAPER TAPE FORMAT CODE

Program _____ Prepared by _____

Date _____ Page ____ of ____

* TABS

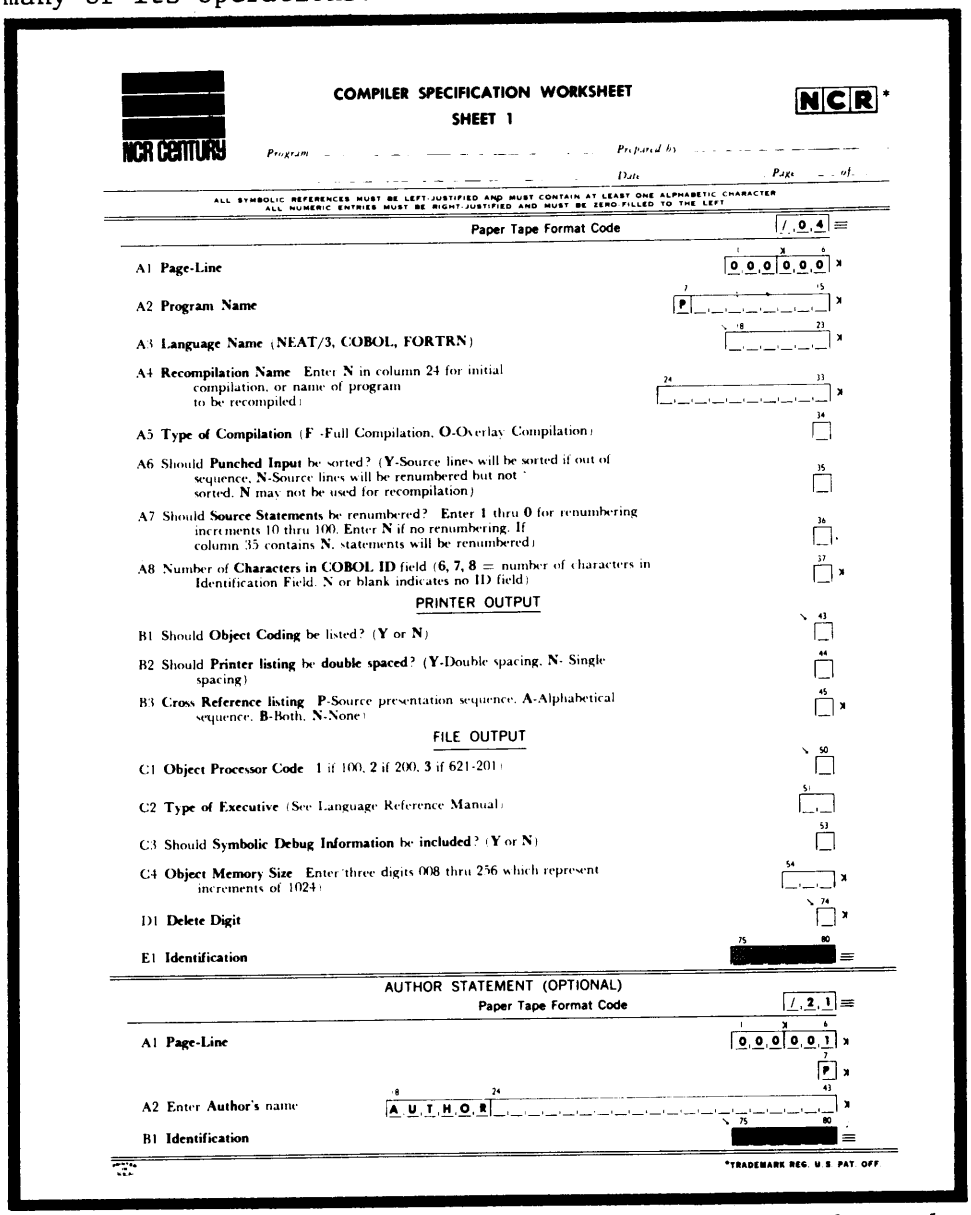
PAGE	LINE		REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE	COMMENTS	IDENTIFICATION
1 2 3	4 5 6	7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74	75 76 77 78 79 80
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									
		D									

The programmer uses the coding worksheets to code his procedural instructions. These procedural instructions may be augmented by major function parameter worksheets which are discussed later.

The programmer uses the data layout worksheets to define the data used in his program as being records, areas, and fields.

Control, File, and Table Specification Worksheets

The control, file, and table specification worksheets ask the programmer questions about his program. The programmer's answers guide the compiler through many of its operations. A control worksheet is illustrated:



NCR CENTURY

COMPILER SPECIFICATION WORKSHEET
SHEET 1

NCR*

Program _____ *Prepared by* _____
Date _____ *Page* ___ *of* ___

ALL SYMBOLIC REFERENCES MUST BE LEFT-JUSTIFIED AND MUST CONTAIN AT LEAST ONE ALPHABETIC CHARACTER
ALL NUMERIC ENTRIES MUST BE RIGHT-JUSTIFIED AND MUST BE ZERO-FILLED TO THE LEFT

Paper Tape Format Code [/,0,4]

A1 **Page-Line** [0,0,0,0,0,0]

A2 **Program Name** [P]

A3 **Language Name** (NEAT/3, COBOL, FORTRN)

A4 **Recompilation Name** Enter **N** in column 24 for initial compilation, or name of program to be recompiled)

A5 **Type of Compilation** (**F**-Full Compilation, **O**-Overlay Compilation)

A6 Should **Punched Input** be sorted? (**Y**-Source lines will be sorted if out of sequence, **N**-Source lines will be renumbered but not sorted. **N** may not be used for recompilation)

A7 Should **Source Statements** be renumbered? Enter 1 thru 0 for renumbering increments 10 thru 100. Enter **N** if no renumbering. If column 35 contains **N**, statements will be renumbered)

A8 Number of **Characters in COBOL ID** field (6, 7, 8 = number of characters in Identification Field. N or blank indicates no ID field)

PRINTER OUTPUT

B1 Should **Object Coding** be listed? (**Y** or **N**)

B2 Should **Printer listing** be **double spaced**? (**Y**-Double spacing, **N**-Single spacing)

B3 **Cross Reference listing** **P**-Source presentation sequence, **A**-Alphabetical sequence, **B**-Both, **N**-None)

FILE OUTPUT

C1 **Object Processor Code** 1 if 100, 2 if 200, 3 if 621-201)

C2 **Type of Executive** (See Language Reference Manual)

C3 Should **Symbolic Debug Information** be **included**? (**Y** or **N**)

C4 **Object Memory Size** Enter three digits 008 thru 256 which represent increments of 1024)

D1 **Delete Digit**

E1 **Identification**

AUTHOR STATEMENT (OPTIONAL)
Paper Tape Format Code [/,2,1]

A1 **Page-Line** [0,0,0,0,0,1]

[P]

A2 Enter **Author's** name [A,U,T,H,O,R]

B1 **Identification**

*TRADEMARK REG. U.S. PAT. OFF

The compiler control worksheet requests basic information about the particular program to be compiled. This information concerns the input of the source program, the compiler treatment of the program, and the output of the object program.

The file specification worksheets request information that the compiler and the I/O Executive need to properly handle the files. This information includes the name of the file, the type of file, the length of each record in the file, etc.

The table specification worksheet requests information which describes the characteristics of a table to be used during program execution. This information includes the name of the table, the maximum number of items in the table, the structure of the table, etc.

Major Functions Parameter Worksheets

The major functions parameter worksheets ask the programmer questions about the function he wishes to perform. From the programmer's answers to the questions asked, the NEAT/3 Compiler develops the necessary object coding to perform the desired function. These major functions reduce the time required to define and implement a program on the NCR Century Series and should be used wherever applicable.

COMMON WORKSHEET RULES

Before the programmer starts to code his program, he should be familiar with some basic coding rules.

- Use a soft-lead pencil to make entries. This produces dark, heavy characters which are easily distinguished by the keypunch operator.
- Be neat, write legibly, and keep each character within its proper location on the worksheet.
- Write questionable characters in a unique manner. For example:
  - Enter the letter O as Θ or Ø to distinguish it from the written numeral 0.
  - Enter the letter Z as Ƶ to distinguish it from the written numeral 2.
  - Enter any lower case letter with a bar over it, such as $\bar{s}$, $\bar{u}$, $\bar{v}$, etc.
  - Enter a space character as ∅.
- Leave every other line blank on the data layout and the coding sheets. This allows the programmer to easily add or change a line, and it also makes the entries easier for the keypunch operator to see.
- After the program is written, scan all source lines for obvious omissions or errors, and make the necessary corrections before the source program is punched.

Maximum Record Size

Punched-card records have a maximum size of 80 characters. (The continuation line is the only exception to this rule. This line is explained later.)

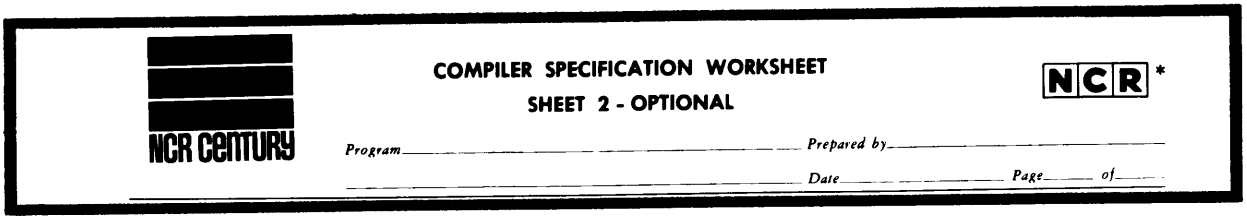
Punched-paper-tape records have a maximum size of 103 characters -- 3 characters for the paper tape format code and a maximum of 100 characters for the source line entries.

COMMON ENTRIES ON PROGRAMMING WORKSHEETS

Many entries on the programming worksheets are common to all worksheets and are discussed here. These common entries are the header, the paper tape format code, the page-and-line number, the worksheet code, the delete digit, and the identification code. The comments are common to the data layout and the coding sheets and are also discussed here.

Header

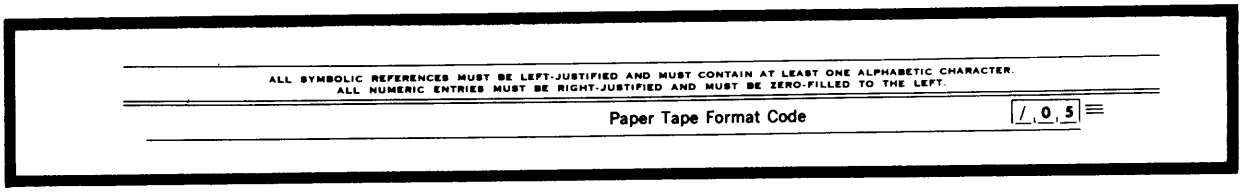
All worksheets have a common header, as illustrated below:



Enter in this header the program name, the programmer's name, the date, the page number of the worksheet, and the number of worksheets used to code the entire program. This header, which is not punched and does not become part of the source lines input to the compiler, aids the programmer both in organizing all worksheets into their proper order (for instance, page 4 of 20) and in documenting the program.

Paper Tape Format Code

All worksheets contain a preprinted paper tape format code which must be punched into certain paper-tape source lines. Every time the source type of the current source line changes from the type of the previous source line, the paper tape format code of the current source line must be punched. The compiler assumes that a source line without a punched paper tape format code is of the same type as the previous source line.



Page-and-Line Numbers

The page-and-line numbers in a source program can be used for three optional precompilation functions. Before the program is compiled, the Source Program Utility Routines can do the following:

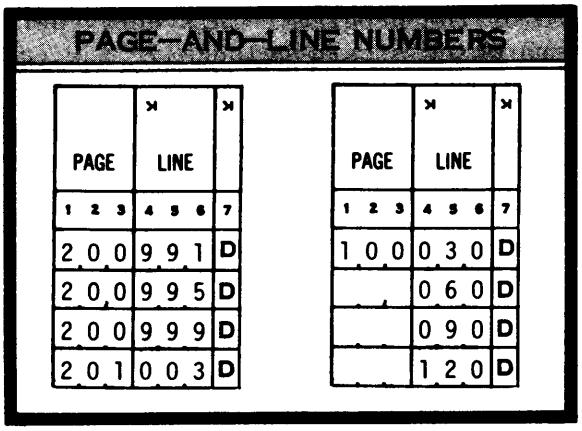
- Sort the source lines in a program into their proper page-and-line number sequence.
- Omit a source line or range of lines from the program.
- Copy a line or range of lines from a compiled program into a specific location of the source program.

Enter in each source line the page-and-line number of the source statement. This number may contain any combination of numeric (0-9) characters. Assign to each source statement a line number in ascending sequence within the page number. Zero-fill these numbers to the left.

It is advisable to leave a gap of 30 consecutive line numbers between each source statement. This permits the programmer to insert additional source statements (for instance, when he debugs the program) without renumbering the ordered statements.

Line numbers in increments of 30 are preprinted on one side of the data layout and the coding sheets. If the programmer wishes to code on this side, he need only enter the page number. The line numbers on the reverse side of these worksheets, however, are left blank. The programmer may wish to use this side to insert additional lines of coding where the consecutive line numbers are not 30 numbers apart. However, the programmer should not code on both sides of the same sheet.

The following examples illustrate correctly sequenced page-and-line numbers:



Worksheet Code

Position 7 of every line on all worksheets contains a preprinted, 1-character code which identifies the worksheet upon which the source statement is coded. This code must be punched as part of each source line.

These worksheet codes and their respective worksheets are:

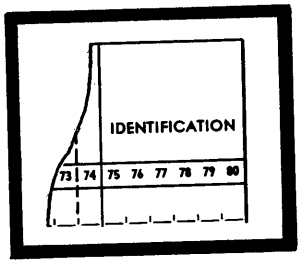
C Coding	M Major Functions
D Data Layout	P Compiler Control
F File Specification	T Table Specification

Delete Digit

Position 74 on all worksheets is reserved for the Delete Digit. See INSTRUCTIONS tab 3, "Compiler Control Worksheet" for an explanation of the Delete Digit.

Identification

The identification code, which may be entered if the source program is on punched cards, occupies positions 75-80.



If the programmer elects to use this option, he should enter the same 6-character tag on every card of the program. This entry permits visual identification of the program to which a card belongs if the card should happen to be misplaced. Any character in the NCR Century Code Chart may be used. The compiler prints this identification field on the program listing.

Comments

The coding and the data layout sheets may contain information entered as comments. Comments are remarks that the programmer makes to document his program. In long programs, comments help the programmer locate a specific area of his program for checking. Comments may contain any character in the NCR Century Code Chart, including the space.

The comments are punched as part of the source program and are included in the source program printout from the compiler. However, these comments do not become part of the object program.

- Partial-Line Comment

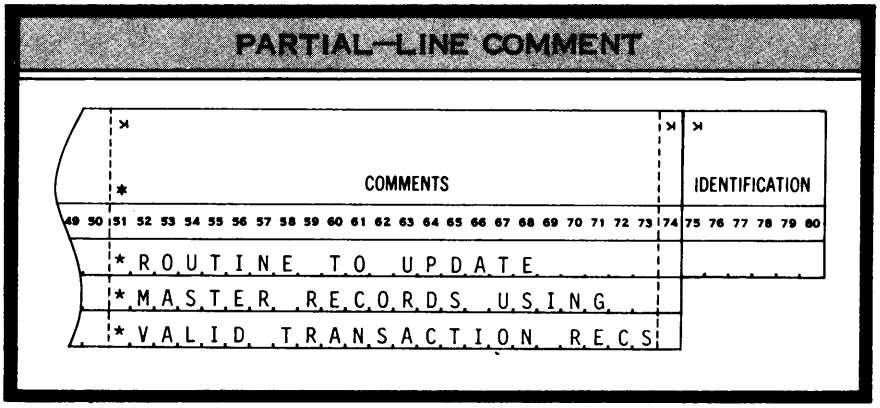
Enter an asterisk (*) in position 31 or beyond of any source statement on the data layout and the coding sheets to signify the beginning of a comment. Then enter the comment.

The length of the source statement, including the comment, must not extend beyond the maximum length specified for the input media (i.e., 73 characters for punched cards, and 100 source characters for punched paper tape).

The extra length permitted on punched-paper-tape source lines is to accommodate long comments, because only a comment may extend beyond position 73.

Most programmers elect to begin all or most of their comments along a predetermined margin. A broken line is printed between positions 50 and 51 for this purpose. Hence, the programmer may enter his comments beginning with an asterisk in column 51. Then, when he receives the program listing, all the comments are neatly aligned on the printed page. This allows him to quickly scan the comments.

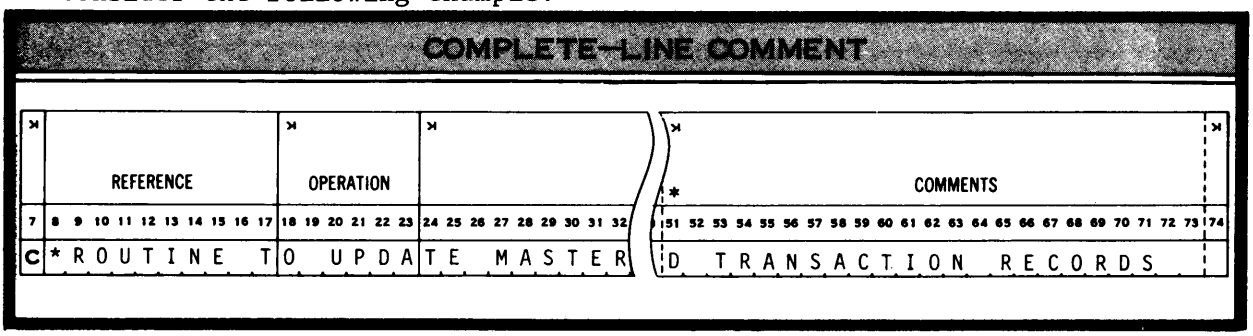
Consider the following example:



- Complete-Line Comment

Enter an asterisk in position 8 of any source line on the coding and the data layout sheets to signify that the entire line, up to and including position 73, is a comment and is not to become part of the object program.

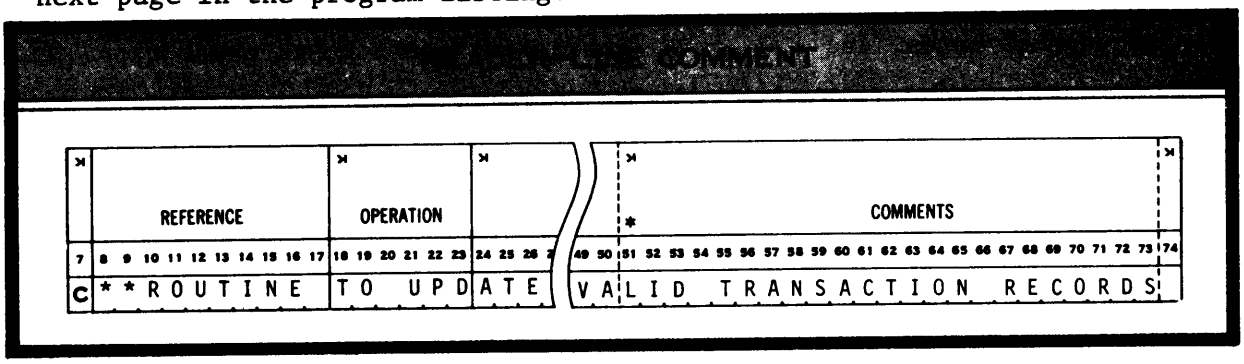
Consider the following example:



- Header-Line Comment

  Enter an asterisk in positions 8 and 9 of any source line on the coding or data layout sheet to signify to the compiler that this line is to printed at the top of a new page in the program listing.

  In the following example, the compiler prints the line at the top of the next page in the program listing.



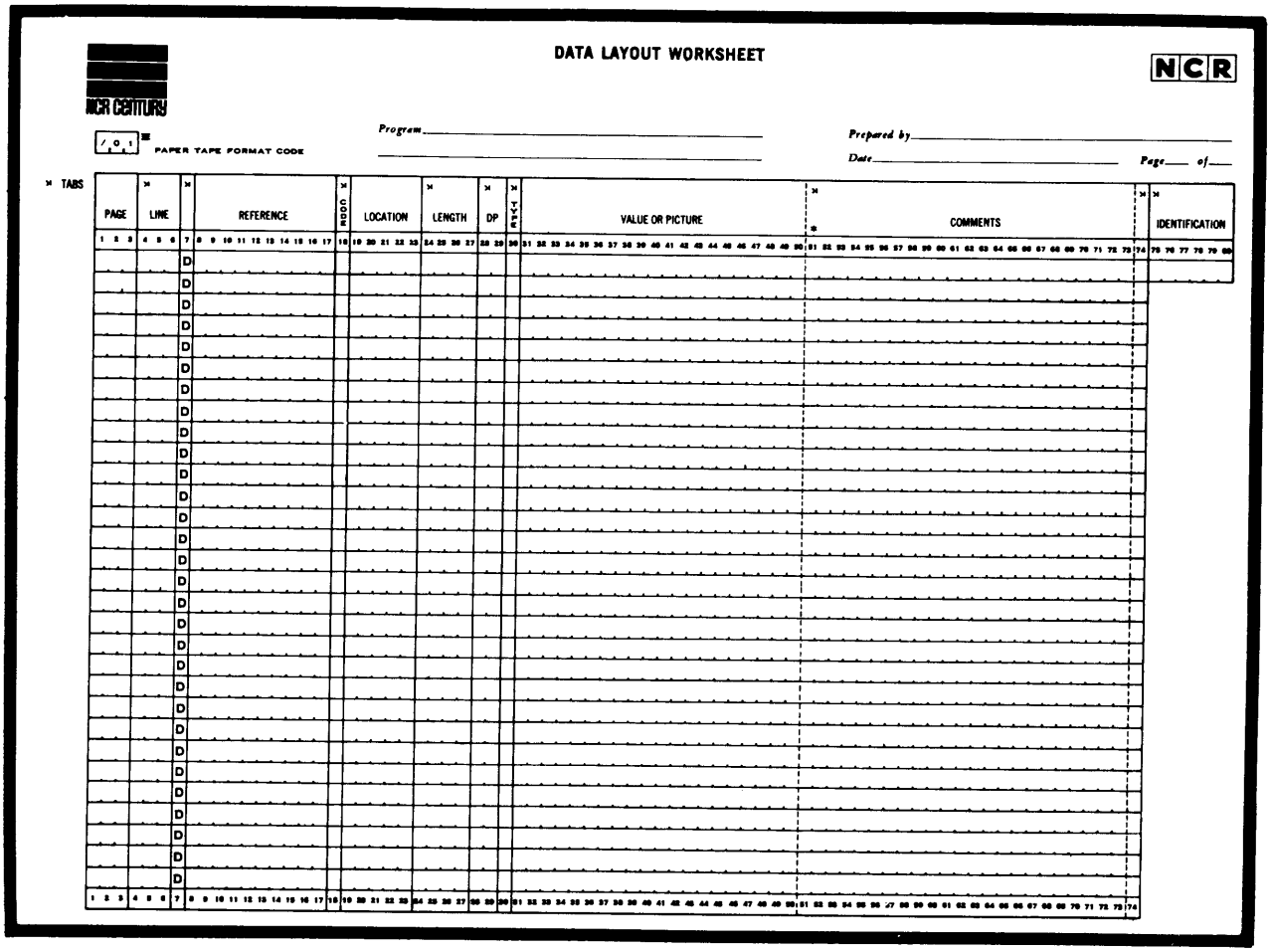
* * * *

DATA LAYOUT SHEETS

ENTRIES ON DATA LAYOUT SHEETS

Data layout sheets for the NEAT/3 Compiler are the worksheets upon which the programmer defines the data to be processed as being records, fields, and areas. The compiler accepts the information entered on these sheets, allocates memory space, assigns addresses, and builds constants.

A data layout sheet is illustrated. Note that a maximum of 30 data definitions can be coded on each sheet, one definition for each line. Note, also, that each line is divided into nine major fields which aid the programmer in defining the data.



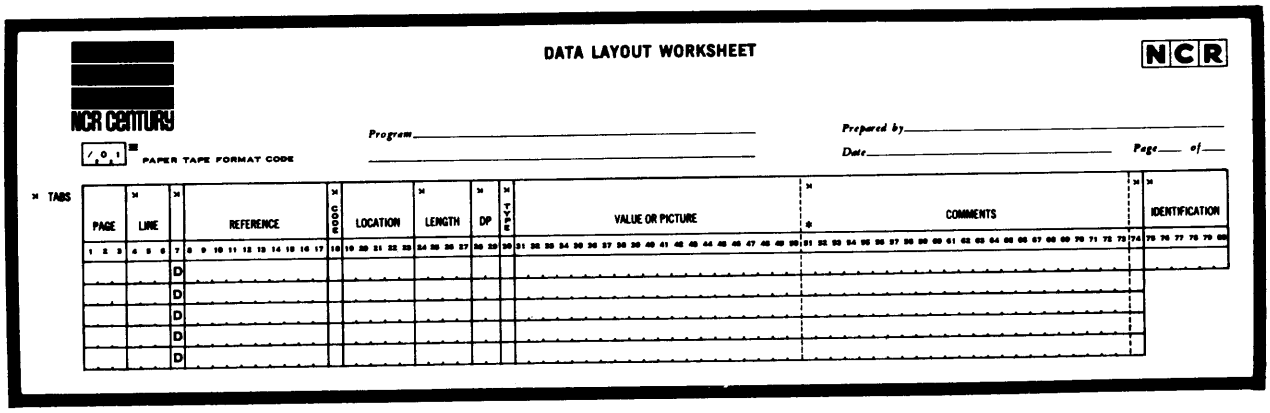
To illustrate the entries required for each of these major fields, the following record named SALESREC will be defined. This record is the daily sales report that each store in a grocery chain sends to the chain headquarters. The carets (^) indicate an implied decimal-point position. Each x indicates a character position.

FORMAT OF SALESREC

Store Code	Manager	Date			Grocery	Produce	Meat	Dairy	Misc.	Daily Total
		Da	Mo	Yr						
xxx	xxxxxxxx	xx	xx	xx	xxxx^xx	xxx^xx	xxxx^xx	xxx^xx	xxx^xx	xxxx^xx

Common Entries

The header, the paper tape format code, the page-and-line number (positions 1-6), the worksheet code (position 7), the comments, the delete digit (position 74), and the identification tag (positions 75-80) are defined under INTRODUCTION AND DATA, tab 3, "Programming Worksheets."



Reference

A programmer may enter in positions 8-17 of the data layout sheet a symbolic reference tag to identify the record, area, or field being defined. (A reference tag is not needed if the record, area, or field is not accessed during processing.)

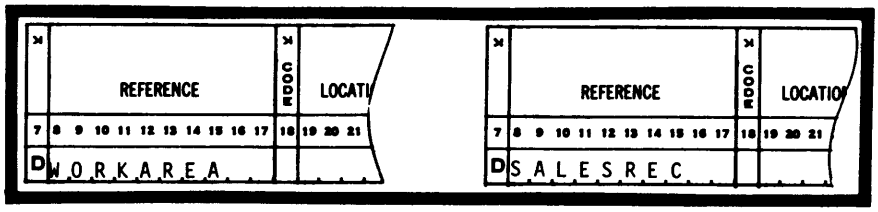
Usually, the programmer enters a near-English word which allows him to immediately identify the data or the area to which it refers. For instance, a programmer may reference the data definition of a transaction record by SALESREC and the definition of a work area by WORKAREA.

The reference tag may contain from 1 to 10 characters which are made up of the alphabet (A-Z) and/or the numerals (0-9). Each tag must begin in position 8 and must contain at least one alphabetical character. Spaces are not permitted within the tag.

Unless it is used as a qualifier (see INTRODUCTION AND DATA, tab 3, "Coding Sheets"), each tag must be unique to the program; i.e., it may appear as a reference tag in only one source statement in the program. However, the name may appear as an operand as often as necessary.

The compiler, as it processes each source statement, checks positions 8-17 for an entry. If it finds a reference tag, the compiler associates with this reference tag all the information pertinent to where the data or area is stored.

The following example illustrates correctly entered reference tags:



Code

Enter in position 18 of each line a 1-character code to indicate that the data being defined is either a record (R), an area (A), or a field (F).

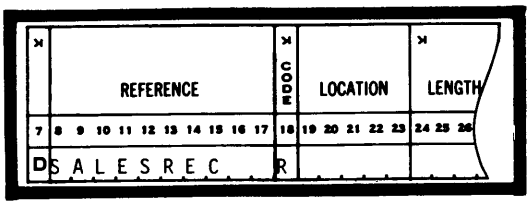
- Record Code

  Enter R if the data being defined is a record. The compiler treats the record definition as a blueprint of the format of records within a particular file.

  Records may be fixed or variable in length. If all the records in a file contain the same number of characters, the records are fixed in length. If, however, the number of characters differs among the records in the same file, these records are variable in length.

  The record definition does not reserve memory space. When input to the compiler, the record definition must immediately follow its associated file specification statements. These file specification statements reserve a memory space called a buffer area. During the production run, the input command accesses a predetermined number of records -- a block of records -- and stores this block in the buffer area. Likewise, the output command accesses a block of updated records and stores this block on an external storage media.

  The following example illustrates the R code entry.



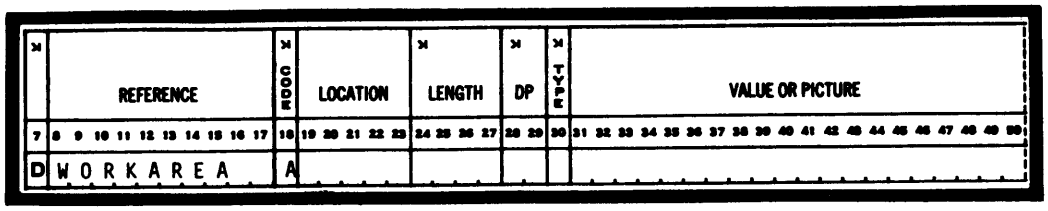
- Area Code

  Enter A if the data being defined is in an area. Since an area is not associated with a file, the compiler allocates for each area the amount of memory space specified in that area definition. Each area definition may serve one of two purposes:

  1. It may simply reserve memory space (working-storage area). Values may be temporarily stored within this area during program execution.

  2. It may both reserve memory space and fill it with constants (or unchanging values which the program may repeatedly need during program execution).

  The following example illustrates the use of the A code.



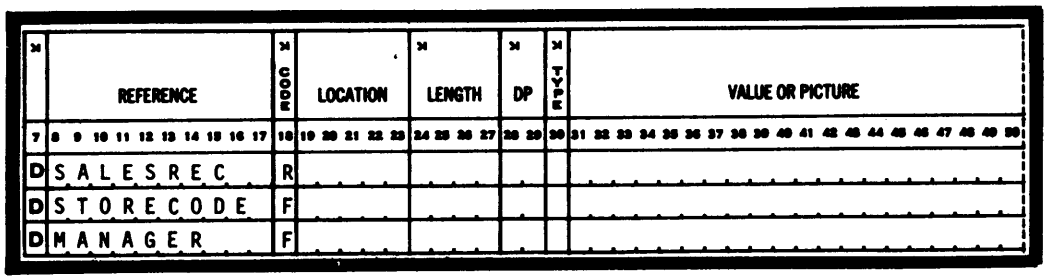
	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	WORKAREA	A					

- Field Code

  Enter F if the data being defined is in a field. A field is a subdivision of either a record or an area. Defining the field permits the programmer to access portions of records or areas.

  The data definition of a field must immediately follow its associated record or area definition. (Intervening record or area definitions may not separate field definitions from their own record or area definition.)

  The following example defines two fields in the grocery-chain record SALESREC.

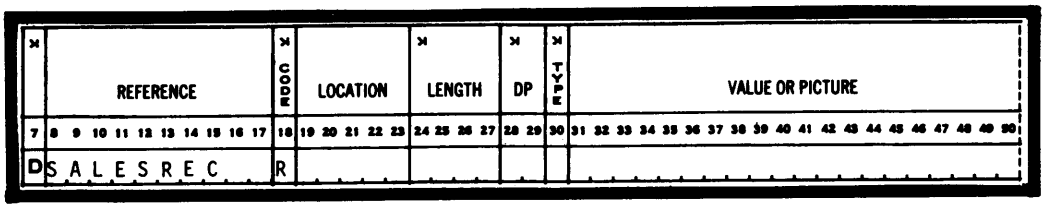


	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	SALESREC	R					
D	STORECODE	F					
D	MANAGER	F					

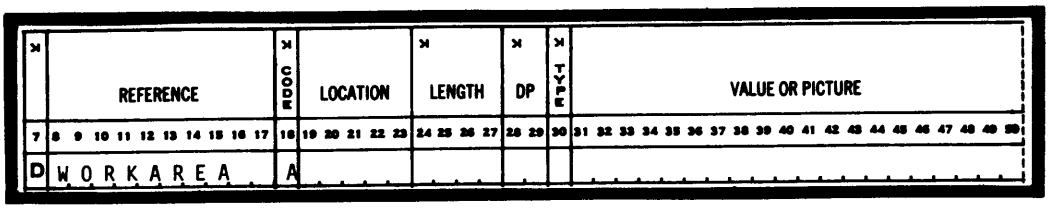
Location

- Location Entry for Record and Area Definitions

Normally, leave positions 19-23 (LOCATION) blank for record and area definitions. (See below for the exceptions.) When these positions for a record definition are left blank, the compiler associates the format and the name of this record with the buffer area of its associated file. Consider the following definition:



When the location positions for an area definition are left blank, the compiler allocates space for the area and, if specified, places a constant within it. Consider the following definition:



- Exception 1

Positions 19-23 (LOCATION) for record and area definitions may contain SAME☑.

If the formats and field names of two or more records of fixed or variable length are to be associated with the same file, define each record and its fields as usual; but enter SAME☑ in the location columns of the second and succeeding record definitions. The compiler then assigns each succeeding record definition to the same buffer area associated with the first record definition.

If the records are variable in length, the first record definition encountered by the compiler must be that of the longest record.

For instance, a file containing information about the parts manufactured by a certain company has fixed-length records. Each record is in either of two formats. The programmer defines these record formats as:

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	PARTSRECA	R		38			
D	KEY	F	0	1			
D	~~						
D	format A						
D	~~						
D							
D	PARTSRECB	R	SAME	38			
D	KEY	F	0	1			
D	~~						
D							
D	format B						
D	~~						
D							

During processing, the program only has to look at the contents of KEY to determine if the record is in Format A or in Format B. It can then reference the fields within the appropriate record format.

Likewise, if the formats and field names of two or more areas are to be associated with the same area in memory, define each area and its fields as usual, but with the following exceptions:

1. Do not enter any values in the value positions;
2. Enter SAME in the location columns of the second and succeeding area definitions.

The compiler then assigns each succeeding area definition to the same address associated with the first area definition.

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	WORKSTRG1	A		12			
D	RATE	F	0	3	3	U	
D	BALANCE	F	3	9	2	U	
D							
D	WORKSTRG2	A	SAME	12			
D	TOTAL	F	0	10	2	U	
D	DISCOUNT	F	10	2	2	U	

** • Exception 2

SAME⍁ entered in positions 19-23 may also allow two files to share the same buffer area. However, the files must be stored on the same external media; and, although in the same program, they must never be processed at the same time during program execution.

For example, PROGRAM1 contains eight files -- four of which are never being processed at the same time. These four files are all stored on disc. The size of the buffer area required for FILE1 is 500 characters, for FILE2 is 450 characters, for FILE3 is 512 characters, and for FILE4 is 500 characters.

If each of these four files has its own buffer area, 1962 characters of memory would be reserved at all times during program execution; however, only a maximum of 512 characters would be utilized at any one time.

Therefore, as one way to conserve memory space, the programmer may define these files to share the same buffer area. Follow these rules:

1. Define on file specification sheets the file requiring the largest buffer area (FILE3 in the above example). The first file defined must require not only the largest buffer, but the largest number of buffers as well.

2. Define on data layout sheets the format of the records in this file. Leave blank the location positions of the R (record) statement.

3. Define on file specification sheets any other file that is to share the same buffer area as the preceding file's.

4. Define on data layout sheets the format of the records in this file. Enter SAME⍁ in the location positions of the R (record) statement. The compiler, when presented with this sequence of statements, associates the buffer area of this file with the buffer area of the preceding file.

5. Follow steps 3 and 4 as often as necessary. The buffer area of each subsequent file whose R (record) statement contains SAME⍁ in its location positions is associated with the buffer area of the first file in this string.

6. Be sure that these steps are consecutively followed and that the compiler is not presented with an intervening definition of a file that is not to share the same buffer area as the others.

When the files in PROGRAM1 are defined, the source lines for the four files that share the same buffer area are presented to the compiler in the following order:

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	*FILE SPEC		FOR FILE3			GO	ES HERE
D	RECORD3	R		32			*16 RECS IN 1 BUFFER
D	( )						*REQUIRES A 512-
D	format 3						*CHARACTER BUFFER
D	( )						
D	( )						
D							
D	*FILE SPEC		FOR FILE1			GO	ES HERE
D	RECORD1	R	SAME	50			*10 RECS IN 1 BUFFER
D	( )						*REQUIRES A 500-
D	( )						*CHARACTER BUFFER
D	format 1						
D	( )						
D	( )						
D	*FILE SPEC		FOR FILE2			GO	ES HERE
D	RECORD2	R	SAME	45			*10 RECS IN 1 BUFFER
D	( )						*REQUIRES A 450-
D	format 2						*CHARACTER BUFFER
D							
D	( )						
D	( )						
D							
D	*FILE SPEC		FOR FILE4			GO	ES HERE
D	RECORD4	R	SAME	25			*20 RECS IN 1 BUFFER
D	( )						*REQUIRES A 500-
D	( )						*CHARACTER BUFFER
D	format 4						

When the compiler encounters SAME in the location positions of the definitions for RECORD1, RECORD2, and RECORD4, it assigns to the respective files the same buffer area that was assigned to the file containing RECORD3.

This buffer area is 512 characters long. (This length is specified on the file specification sheet for FILE3.)

- Location Entry for Field Definitions

Enter in positions 19-23 the relative location of the field within the record or the area. This location entry must be right-justified. (The rightmost digit of this entry must occupy position 23.) The position of first character in the record or area is relative location zero; the position of second character is relative location one; etc.

The following example illustrates the relative location of fields within the grocery chain record, SALESREC.

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	SALESREC	R		50			
D	STORECODE	F	0	3			
D	MANAGER	F	3	8			
D	DATE	F	11	6			
D	DAY	F	11	2			
D	MONTH	F	13	2			
D	YEAR	F	15	2			
D	GROCERY	F	17	6			

A location entry is not required when the fields being defined are adjacent. In the example below, the same record (SALESREC) has been described omitting unnecessary location entries.

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	SALESREC	R		50			
D	STORECODE	F		3			
D	MANAGER	F		8			
D	DATE	F		6			
D	DAY	F	11	2			
D	MONTH	F		2			
D	YEAR	F		2			
D	GROCERY	F		6			

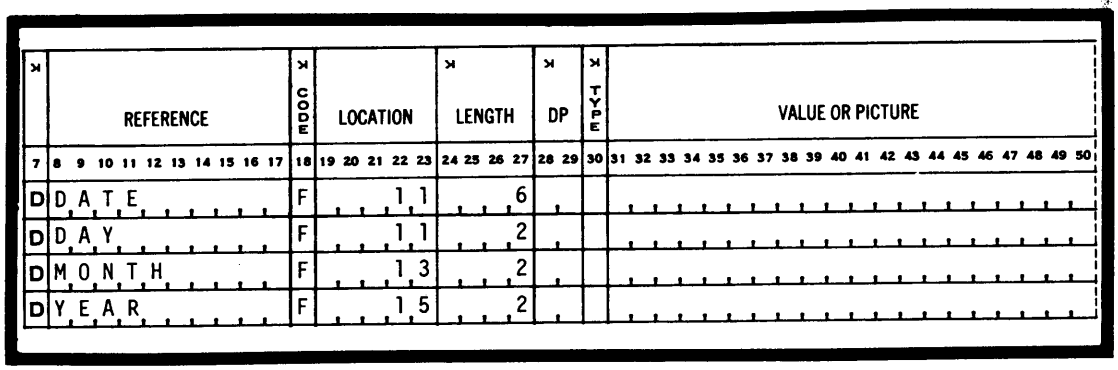
The compiler always associates relative location 0 with the first field. Using the specified length of the first field, the compiler generates the relative location of the second field; using the specified length of the second field, the compiler generates the relative location of the third field.

Because the fourth field (DAY) is not adjacent to the DATE field, the programmer must specify a location. The remaining fields are all adjacent (MONTH is adjacent to DAY, YEAR is adjacent to MONTH, GROCERY is adjacent to YEAR); therefore, the compiler generates the location entries for the last three fields.

Length

Use positions 24-27 to specify the length of the record, area, or field being defined. This entry is right-justified; i.e. the rightmost digit occupies position 27.

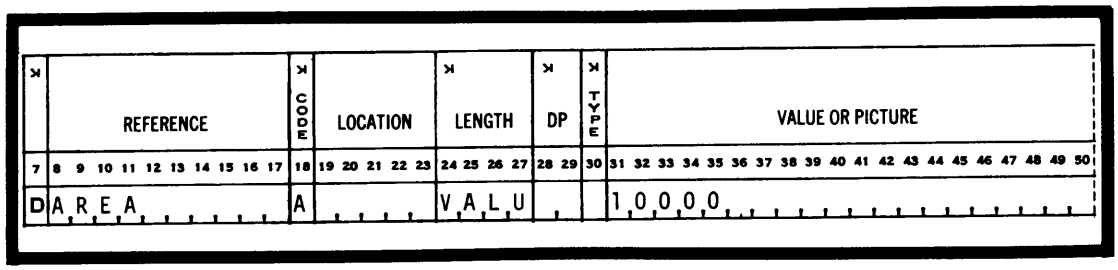
The compiler associates the specified length with the data name in the reference positions. In the following example, the compiler associates two memory positions beginning in relative location 13 with the reference MONTH. (This 2-character field is included in the 6-character field DATE.)



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
	7 8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	DATE	F	11	6			
D	DAY	F	11	2			
D	MONTH	F	13	2			
D	YEAR	F	15	2			

When variable-length records are being processed, remember to enter as the length the length of the largest record.

The maximum number that could be entered in this 4-character position is 9999. If the length entry is greater than 9999, the word VALU is entered in positions 24-27; the actual length is specified in position 31. Consider the example below.



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
	7 8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	AREA	A		VALU			10000

Note: A VALU entry is restricted to an area definition that does not have an entry in the value or picture positions. The maximum length that can be used with VALU is 65,536.

For further explanation on the length entry, refer to the specific data-type descriptions in this publicaiton.

DP (Decimal Point)

Positions 28 and 29 are reserved for a DP entry. This entry, which may range from 0-99, indicates the number of decimal positions in a data item.

The type of data being defined determines whether or not a DP entry is required. For example, alphanumeric (X-type) data may not have a DP entry; an editing mask (E-type data) requires a DP entry when it defines a numeric mask, but not when it defines an alphanumeric mask; U-, D-, P-, and K-type data require a DP entry if the field does not consist entirely of whole integers. (For further explanation on when to enter a DP value, see the individual data-type descriptions in this publication.)

Note from the following examples that the DP entry affects the way the value is stored.

Source Value	DP	Stored Value
543.21	2	543^21
543.21	3	543^210

Type

Enter in position 30 (TYPE) of every field definition a 1-character code. This code specifies the type of the data being defined, whether this data is input from file records, whether it is specified on a data layout sheet as a constant, or whether it is moved into this field during processing.

The valid codes and their corresponding data types follow:

Code	Data Type
☑ or X	Alphanumeric Characters
S	Generated Spaces
Z	Generated Zeros
U	Unsigned Decimal
D	Signed Decimal
B	Binary
H	Hexadecimal
P	Signed Packed Decimal
K	Unsigned Packed Decimal
E	Editing Mask

An entry in the TYPE position is optional for the definition of a record or an area. If all the data in the record or area is of the same type, enter the appropriate code. For instance, if all the fields within a record or an area contain packed data, define the type code as K. If the data is of many types, either leave this position blank or define the type code as X. (See the Data Description section of this publication for more information about these codes.)

Up to this point, only excerpts of the data definitions for SALESREC have been illustrated. It is now possible to understand the complete definitions for SALESREC and its associated fields.

The format of SALESREC follows:

FORMAT OF SALESREC

Store Code	Manager	Date			Grocery	Produce	Meat	Dairy	Misc.	Daily Total
		Da	Mo	Yr						
xxx	xxxxxxxx	xx	xx	xx	xxxx^xx	xxx^xx	xxxx^xx	xxx^xx	xxx^xx	xxxx^xx

Using the above format, the programmer defines SALESREC as follows:

	REFERENCE	RECORD	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	SALESREC	R		50			
D	STORECODE	F	0	3		X	
D	MANAGER	F	3	8		X	
D	DATE	F	11	6		X	
D	DAY	F	11	2		X	
D	MONTH	F	13	2		X	
D	YEAR	F	15	2		X	
D	GROCERY	F	17	6	02	U	
D	PRODUCE	F	23	5	02	U	
D	MEAT	F	28	6	02	U	
D	DAIRY	F	34	5	02	U	
D	MISC	F	39	5	02	U	
D	DAILYTOTAL	F	44	6	02	U	

Value or Picture

The entry starting in position 31 varies with the record, area, or field being defined. The programmer may leave these positions blank, or he may enter either a value or a picture.

- When to Leave These Positions Blank

  Leave these value or picture positions blank in all record definitions and in those area definitions that allocate memory space but that do not require an initial setting at compilation time. Consider the following definitions:

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	SALESREC	R		50		X	
D	WORKAREA	A		18	02	U	

- When to Enter a Value

  Enter a value if the source line is an area or field definition of a constant. This value must not extend beyond position 73. (Position 74 is reserved on all worksheets for the delete digit. See compiler control sheet, position 74.) Consider the following definitions:

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	MULTIPLIER	A		5	4	U	3.1416
D	KEYS	A		6			
D	KEY1	F	0	2		U	1.0
D	KEY2	F	2	2		U	2.0
D	KEY3	F	4	2		U	3.0

- When to Enter a Picture

  Enter a picture if the source line specifies a picture of how data is to be edited before it is printed. The mask must not extend beyond column 73. The editing mask is explained in greater detail under E TYPE in the following section.

* * * *

DATA DESCRIPTION

Each section of the data layout sheet and all the rules and entry variations that pertain to it have been presented; that is, all the rules that apply to length entries, to DP entries, etc. have been discussed. Now, the data layout sheet will be approached from a different viewpoint. One line of data will be defined at a time, and only those rules that apply to the data will be considered. To do this, assume that the data is of a particular type. The following discussion briefly describes each type of data and reviews all the associated data layout sheet entries that are needed to define this data properly.

X Type

- Data Type

  Alphanumeric Characters.

  The data is made up of any of the 8-bit alphanumeric characters in rows 2 to 5 of the Century Code Chart.

- Use When

  The data in the record, area, or field being defined is in 8-bit alphanumeric characters.

  - Example

    Since the records within a name and address file contain alphabetical and numerical characters, a programmer may define these name and address fields as being X type.

    A programmer may define a header for a report as being X type because the header is an alphanumeric constant.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters (L) that are to be reserved for this data string. If this definition contains an entry in the value positions, the maximum length that can be entered is 43.

- Location

  For a record or an area definition, leave these positions blank.

  For a field definition, enter the relative location of this field within its associated area or record.

- DP

  Leave these positions blank. Alphanumeric characters, by their nature, do not contain a decimal point.

- Value

  If this definition is of a record or of a field within a record, a value entry is invalid.

  If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 8-bit alphanumeric characters and stores the value in the constants section of the object program.

  The compiler checks the first L characters. If they are all blank, the compiler reserves memory space into which a value can later be stored.

  Leading or embedded space characters are valid.

- Comments

  A comment must not begin in any position that the length entry has reserved for this data string; i.e., any comment preceded by an asterisk must not begin before position (31 + L).

- Examples

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE	COMMENTS
D	HEADER	A		17		X	1966 SALES REPORT	

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE	COMMENTS
D	PAGENUMBER	A		8				* PAGE NUMBER - TO BE
D	PAGE	F	0	5		X	PAGE	* INCREMENTED & PRINTED
D	NUMBER	F	5	3		U	001	* AT TOP OF NEW PAGES

U Type

- Data Type

  Unsigned Decimal.

  The data is made up of any combination of the 8-bit numeric characters (0-9) as shown in the NCR Century Code Chart.

- Use When

  The data in the record, area, or field being defined is in 8-bit numeric unsigned decimal characters.

  - Example

    A programmer knows that during processing, the field called AMTSOLD will always contain a positive number. He may, therefore, define this field as being U type.

    A programmer wishes to enter as constants a string of interest rates -- .04, .045, .05, .06, etc. Since these constants are always positive and are numeric, the programmer may define them as being U type.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters (L) that are to be reserved for this data string. Do not count the decimal point as a character in this length. If this definition contains an entry in the value positions, the maximum length that can be entered is 43.

  - Location

    For a record or area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area or record.

  - DP

    Enter the number of 8-bit characters occupying decimal positions in the value being defined.

  - Value

    If this definition is of a record or of a field within a record, a value entry is invalid.

If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 8-bit numeric unsigned decimal characters and stores the value in the constants section of the object program.

The value, if any, must start in position 31. If position 31 is left blank or contains an asterisk (*), the compiler does not look at the remaining positions for a value but reserves memory space into which a value can later be stored.

If the value is an integer, a decimal point need not be entered. Otherwise, the value must contain a decimal point. However, the decimal point is not stored. Leading or trailing zeros are added to store the value properly.

For example, L = 5
DP = 2

The value 12 is stored as 01200
The value 1.2 is stored as 00120
The value .12 is stored as 00012

A space character may not be embedded within the value. If it is, the compiler considers only the characters to the left of the space as the value.

- Comments

Comments, if any, may begin with an asterisk in position 31 or beyond (if no value is entered) or in the first available position (if a value is entered).

- Examples

*	REFERENCE	* CODE	LOCATION	LENGTH	* DP	* TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	INTEREST	A		4		U	
D	6PERCENT	F	0	2	2	U	.06
D	4PERCENT	F	2	2	2	U	.04

*	REFERENCE	* CODE	LOCATION	LENGTH	* DP	* TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	TALLY	A		6	2	U	1 *TALLY COUNT

D Type

- Data Type

  Signed Decimal.

  The data is made up of any combination of the 8-bit numeric characters (0-9) whose positive or negative value is expressed with an 8-bit sign (+ or -). The bit configuration of these characters is shown in the NCR Century Code Chart.

- Use When

  The data in the record, area, or field being defined is in 8-bit signed decimal characters.

  - Example

    During the processing of PROGRAMA, the field BALANCE may sometimes have a positive and sometimes a negative value. The programmer may define this field as being D type.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters that are to be reserved for this data string. Count the sign as part of the length, but not the decimal point. If this definition contains an entry in the value positions, the maximum length that can be entered is 43. In all cases, the minimum length is 2.

  - Location

    For a record or an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area or record.

  - DP

    Enter the number of 8-bit characters occupying decimal positions in the data string being defined.

  - Value

    If this definition is of a record or of a field within a record, a value entry is invalid

    If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 8-bit numeric signed decimal characters and stores the value in the constants section of the object program.

The value, if any, must start in position 31. If position 31 is left blank or contains an asterisk (*), the compiler does not look at the remaining positions but reserves memory space into which a value can later be stored.

If the value is negative, the minus sign must be entered. If the value is positive, the plus sign need not be entered. The compiler generates a plus sign if none is present.

If the value is an integer, a decimal point need not be entered. Otherwise, the source value must contain a decimal point. However, the decimal point is not stored. Leading or trailing zeros are added to store the value properly.

For example, L = 5
DP = 2

The value +12 is stored as 1200+
The value -1.2 is stored as 0120-
The value +.12 is stored as 0012+
The value -123.45 is stored as 2345- (error comment)
The value -1234 is stored as 3400- (error comment)

NOTE: In memory the sign is stored in the rightmost character position of the value.

A space character may not be embedded within the value. If it is, the compiler considers only the characters to the left of the space as the value.

- Comments

Comments, if any, may begin with an asterisk in position 31 (if no value is entered) or in the first available position (if a value is entered).

- Examples

*	REFERENCE	* CODE	LOCATION	* LENGTH	* DP	* TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	CONSTANT	A		2	0	D	-5

*	REFERENCE	* CODE	LOCATION	* LENGTH	* DP	* TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	MULTIPLIER	A		7	3	D	+491.664

K Type

- Data Type

  Unsigned Packed Decimal.

  The data is made up of any combination of the 4-bit numeric characters (0-9) as shown in the NCR Century Code Chart.

- Use When

  The data in the record, area, or field being defined is in 4-bit numeric unsigned packed decimal characters.

  - Example

    Since unsigned packed decimal characters are recorded in less space on a storage device than are unsigned decimal characters, a programmer may choose to store as unsigned packed decimal characters the numeric fields in his file's records.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters that are to be reserved for this data string. Do not count the decimal point as part of this length. Remember that two unsigned packed decimal characters are stored in one memory position; therefore, if this definition contains an entry in the value positions, the maximum length that can be entered is 21.

  - Location

    For a record or an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area or record.

  - DP

    Enter the number of 4-bit characters occupying decimal positions in the data string being defined.

  - Value

    If this definition is of a record or of a field within a record, a value entry is invalid.

If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 4-bit unsigned packed decimal characters and stores this value into the constants section of the object program.

The value, if any, must start in position 31. If position 31 is left blank or contains an asterisk (*), the compiler does not look at the remaining positions for a value but reserves memory space into which a value can later be stored.

If the entered value is an integer, a decimal point need not be entered. Otherwise, the entered value must contain a decimal point. However, the decimal point is not stored. Leading or trailing zeros are added to store the value properly.

For example, L = 4
DP = 3

The value 54321 is stored as 54 32 10 00
The value 5432.1 is stored as 05 43 21 00

A space character may not be embedded within the value. If it is, the compiler considers only the characters to the left of the space as the value.

- Comments

  Comments, if any, may begin with an asterisk in position 31 (if no value is entered) or in the first available position (if a value is entered).

- Examples

*	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	MULTIPLIER	A		10		K	
D	MULTA	F	0	5	0	K	9 8 7 6 5 4 3 2 1 0
D	MULTB	F	5	5	3	K	9 8 7 6 5 4 3 . 2 1 0

P Type

- Data Type

  Signed Packed Decimal.

  The data is made up of any combination of the 4-bit numeric characters (0-9) whose positive or negative value is expressed with a 4-bit sign (+ or -). The bit configuration of these characters is shown in the NCR Century Code Chart.

- Use When

  The data in the record, field, or area being defined is in 4-bit signed packed decimal characters.

  - Example

    Since signed packed decimal characters are recorded in less space on a storage device than are signed decimal characters, a programmer may choose to store as signed packed decimal characters the signed numerical fields in his file's records.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters that are to be reserved for this data string. Count the sign as part of the length, but not the decimal point. Remember that two signed packed decimal characters are stored in one memory position; therefore, if this definition contains an entry in the value positions, the maximum length that can be entered is 21.

  - Location

    For a record or an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area or record.

  - DP

    Enter the number of 4-bit characters occupying decimal positions in the data string being defined.

  - Value

    If this definition is of a record or of a field within a record, a value entry is invalid.

If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 4-bit signed packed decimal characters and stores this value into the constants section of the object program.

The value, if any, must start in position 31 with a plus or a minus sign. If position 31 is left blank or contains an asterisk (*), the compiler does not look at the remaining positions for a value but reserves memory space into which a value can later be stored.

If the entered value is an integer, a decimal point need not be entered. Otherwise, the entered value must contain a decimal point. However, the decimal point is not stored. Leading or trailing zeros are added to store the value properly.

For example, L = 4
DP = 3

The value -5432.1 is stored as 54 32 10 0-
The value +54.321 is stored as 00 54 32 1+

NOTE: In memory the sign is stored in the rightmost character position of the value.

A space character may not be embedded within the value. If it is, the compiler considers only the characters to the left of the space as the value.

- Comments

Comments, if any, may begin with an asterisk in position 31 (if no value is entered) or in the first available position (if a value is entered).

- Examples

	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
D	DIVIDER	A		10		P	
D	DIVIDEA	F	0	5	4	P	-98765.4321
D	DIVIDEB	F	5	5	7	P	+98.7654321

Z Type

- Data Type

  Generated Zeros.

  The compiler generates 8-bit zeros to fill the specified length. The bit configuration of these generated zeros is shown in the NCR Century Code Chart.

  When the reference of a Z field is used as an operand, the data in this field is treated as unsigned decimal characters (U type).

- Use When

  The area or field being defined is originally to be zero-filled.

  - Example

    The header on each printed page is to contain the name of the report, the date, and the page number. To get a different number on each page, the programmer may initially zero-fill the page field. In the print routine, he may increment the number in this field by one each time a page is printed.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit zeros that the compiler is to generate. A maximum of 9999 zeros can be generated.

  - Location

    For an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area.

  - DP

    Enter the number of 8-bit characters occupying decimal positions in the string of generated zeros.

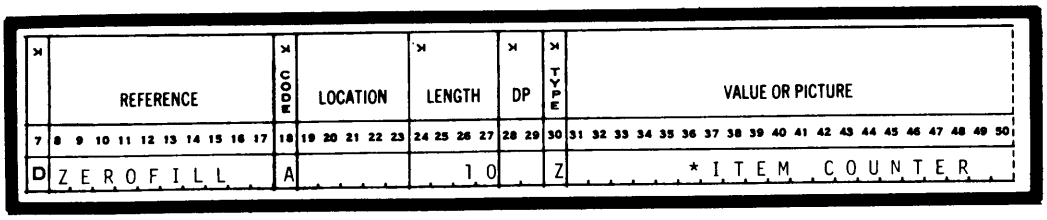
  - Value

    A value entry is invalid.

  - Comments

    Comments, if any, may begin with an asterisk in position 31 or beyond.

- Example



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	ZEROFILL	A		10		Z	*ITEM COUNTER

S Type

- Data Type

  When S-type is used, the compiler fills the area or field being defined with a specified character.

  When the reference of an S field is used as an operand, the data in this field is treated as alphanumeric characters (X type).

- Use When

  The area or field being defined is originally to be filled with a character or spaces.

  - Example

    A programmer may define the header line to be printed at the top of each page in his report as being an area. He may enter the data and the name of the report as constants and space-fill the intervening positions by defining the positions to be in a field of S type.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    The maximum number that can be entered is 9999.

  - Location

    For an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area.

  - DP

    Leave these positions blank.

  - Value

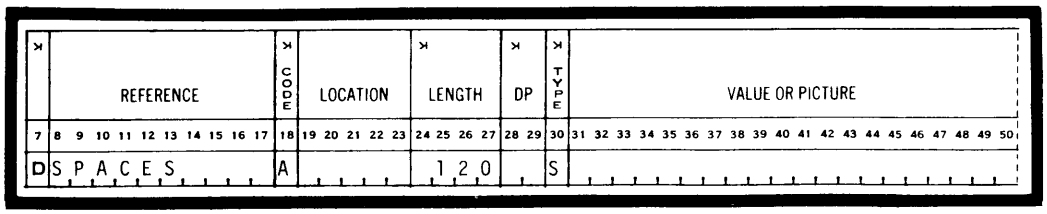
    Positions 31-32 of the value positions are used to indicate the character to be filled. If no character is specified (no entry in positions 31-32), the compiler fills the area or field with spaces.

    When a character is to be filled, enter in positions 31-32 the 2-character hexadecimal representation of the character. The hexadecimal representations of characters can be found on the chart on page 9 of FILES, tab 1, "Paper Tape File Specifications" (Pub. No. 8).

- Comments

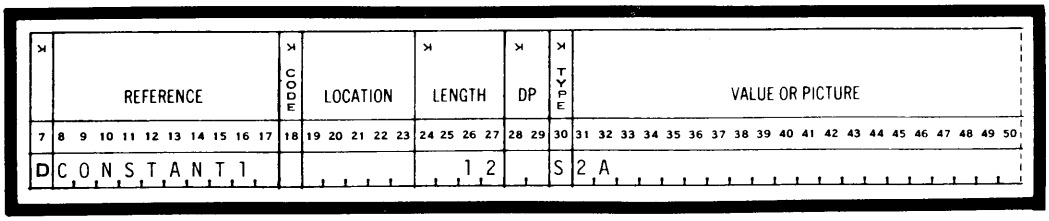
  Comments, if any, may begin with an asterisk in position 31 or beyond.

- Examples



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	SPACES	A		120		S	

In the example below, the 12-character area is filled with asterisks (*).



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	CONSTANT1			12		S	2A

B Type

- Data Type

  Binary.

  The date being defined is a binary number.

- Use When

  The data in the record, area, or field being defined is a binary number.

  - Example

    A programmer wishes to keep a running total of all transactions processed during each run. He chooses to keep this total in a binary field because a large integer value is stored more compactly and thus more efficiently as a binary number than as a decimal number.

- Associated Data Layout Sheet Entries

Any violation of the following conventions will cause an error comment to be generated.

- Length

Enter the number of 8-bit characters that are to be reserved for the binary number. The binary equivalent of any positive integer ranging from 0 to 999,999 is a valid binary number.

The following table shows the number of characters needed to store binary equivalents of positive integers:

Positive Integer	Length of Binary Equivalent
1-255	One 8-bit character
256-65, 535	Two 8-bit characters
65,536-999,999	Three 8-bit characters

A binary field may be defined as up to 9,999 characters in length; however, only the rightmost three characters may contain significant data, as indicated in the above table. The remaining character locations are zero-filled by the compiler.

- Location

For a record or an area definition, leave these positions blank.

For a field definition, enter the relative location of this field within its associated area or record.

- DP

Leave these positions blank. Binary numbers are considered to be unsigned integers.

- Value

  If this definition is of a record or of a field within a record, a value entry is invalid.

  If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into a binary number and stores the value into the constants section of the object program.

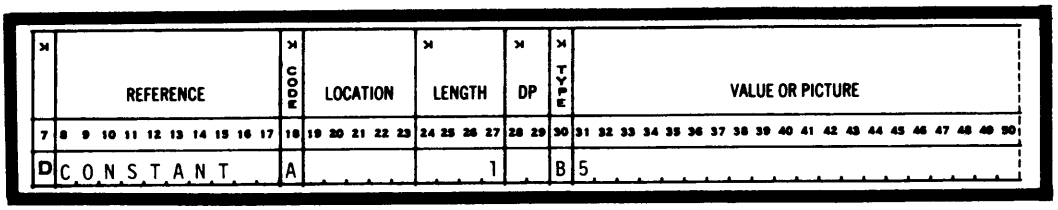
  The value, if any, must start in position 31. If position 31 is left blank or contains as asterisk (*), the compiler does not look at the remaining positions for a value but reserves memory space into which a value can later be stored.

- Comments

  Comments, if any, may begin with an asterisk in position 31 (if no value is entered) or in the first available position (if a value is entered).

- Examples

The compiler translates the decimal value in the following example into a binary five (00000101) and stores this in an area called CONSTANT.



	REFERENCE	CODE	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	CONSTANT	A		1		B	5

H Type

- Data Type

  Hexadecimal.

  The data is the binary representation of the 4-bit hexadecimal characters 0-9 and A-F. This bit configuration is shown in INTRODUCTION AND DATA, tab 2, "Data Concepts."

- Use When

  The data in the area or field being defined is in 4-bit hexadecimal characters.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Reference

    Since hexadecimal data cannot be used as an operand, it should not be given a reference.

  - Location

    For an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area.

  - Length

    Enter the number of 8-bit characters that are to be reserved for this data string. Remember that two hexadecimal characters are stored in one memory position; therefore, if this definition contains an entry in the value positions, the maximum length that can be entered is 21.

  - DP

    Leave these positions blank.

  - Value

    If this definition is of either an area or a field within an area that is to contain a specific value (i.e., a constant), enter this value. The compiler converts this value into 4-bit hexadecimal characters and stores this value into the constants section of the object program.

    The value, if any, must start in position 31. If position 31 is left blank or contains an asterisk (*), the compiler does not look at the remaining positions for a value but reserves memory space into which a value can later be stored.

    A space character may not be embedded within the value. If it is, the compiler considers only the characters to the left of the space as the value.

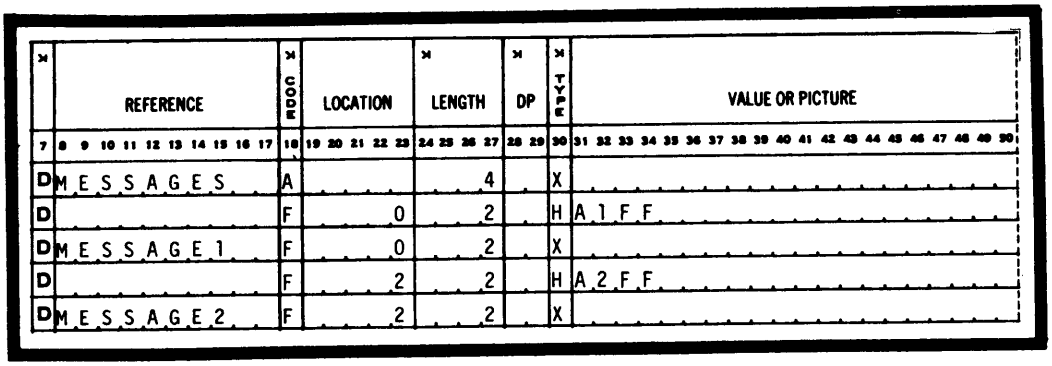
- Comments

  Comments, if any, may begin with an asterisk in position 31 (if no value is entered) or in the first available position (if a value is entered).

- Examples

Since the reference of a field containing hexadecimal data may never be used as an operand, hexadecimal data must be redefined if it is to be accessed during processing.

For instance, an area referenced by MESSAGES is defined as having four alphanumeric 8-bit characters. This area contains two hexadecimal messages -- A1FF and A2FF. These fields are redefined as alphanumeric fields MESSAGE1 and MESSAGE2. Consider the following data definitions:



*	REFERENCE	* CODE	LOCATION	* LENGTH	* DP	* TYPE	VALUE OR PICTURE
7	8 9 10 11 12 13 14 15 16 17	18	19 20 21 22 23	24 25 26 27	28 29	30	31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
D	MESSAGES	A		4		X	
D		F	0	2		H	A1FF
D	MESSAGE1	F	0	2		X	
D		F	2	2		H	A2FF
D	MESSAGE2	F	2	2		X	

When either MESSAGE1 or MESSAGE2 is used as an operand, the data contained in the field is treated as alphanumeric characters and may, therefore, be accessed by procedural instructions.

E Type

- Data Type

  Edited field.

  The E entry in the type position specifies this definition to be of an edited field. The mask itself is entered in the picture positions.

  An editing picture or mask is a string of characters defined over a destination field. This character string describes the format of the data to be output. Generally, only data which is to be printed needs to be edited. After data is moved into a field of E type, the field contains the edited data.

- Use When

  The field being defined is a destination field for data that is to be printed but that first needs special editing to make it more readable.

  - Example

    An insurance company wants a printed report of the total amount of insurance that each salesman has sold during the past month. Since decimal points and currency symbols are not recorded with the data in memory, the programmer may choose to edit all numeric data before it is printed, thereby inserting the currency symbols and the decimal points.

- Associated Data Layout Sheet Entries

  Any violation of the following conventions will cause an error comment to be generated.

  - Length

    Enter the number of 8-bit characters that are in the mask. This length also indicates the length of the field. When edited data is accessed from the field, its length is as specified in this entry.

    The maximum length of a mask is 43 characters. The maximum number of editing characters in the mask for a numeric field -- not including the sign symbols -- is 19 characters. The maximum number of editing characters in the mask for an alphanumeric field is 43 characters. All other characters in the masks are insertion characters. The number of editing characters plus the number of insertion characters in the mask must not exceed 43 characters.

  - Location

    For a record or an area definition, leave these positions blank.

    For a field definition, enter the relative location of this field within its associated area or record.

- DP

  Leave blank the DP positions for the definitions of an alphanumeric field.

  Place an entry in the DP positions for the definition of a numeric field.

  - If decimal positions exist, enter the number of characters to the right of the decimal point which are to be replaced with a data character. This entry does not reflect the character positions needed for insertion characters or for sign characters (+, -, CR, or DB).

  - If no decimal positions exist, enter a zero in position 29 of the data layout sheet.

  NOTE: If the DP positions are left blank, the field is assumed to be alphanumeric.

- Picture

  Enter the editing mask starting in position 31.

  An editing mask (picture) may contain any valid combination of the characters in the Century Code Chart except the space (␣). For a description of each of these characters and the rules governing its use, see the following discussion in this publication entitled EDITING MASKS.

- Example

In the following example, TOTALINE defines the format of the records in a print file. The fields TOTALSALES and TOTRETURNS are destination fields over which an editing mask is defined. Also within this record is a TOTCOMMENT field into which the programmer may have the program store the comment DEPARTMENT TOTALS during program execution. (Ignore for the present time the 4-character TOTCONTROL field because it does not appear in the printed line. This field is a control block; its concept is later explained in conjunction with the printer.)

	REFERENCE	RECORD	LOCATION	LENGTH	DP	TYPE	VALUE OR PICTURE	COMMENTS
D	TOTALINE	R		136				
D	TOTCONTROL	F	0	4		X		
D	TOTCOMMENT	F	4	20		X		
D	TOTALSALES	F	39	14	2	E	ZZZ,ZZZ,ZZZ.XX	*EDITING MASK
D	TOTRETURNS	F	59	14	2	E	ZZZ,ZZZ,ZZZ.XX	*EDITING MASK

* * * *

EDITING MASKS

An editing picture or mask is a string of characters defined over a destination field. This character string describes the format of the data to be output. Generally, only data which is to be printed needs to be edited.

To specify an editing mask for an edited field, the programmer enters on a data layout sheet an E in the type position and the actual mask in the picture positions.

An editing mask (picture) may contain any valid combination of the characters in the NCR Century Code Chart except the space (␣). The rules governing each of these characters differ for masks for alphanumeric fields and for masks for numeric fields.

Masks For Edited Alphanumeric Fields

To edit alphanumeric data, move the data into the destination field over which a mask is defined. A mask for an alphanumeric field is made up of editing characters and insertion characters. The DP positions for the definition of a mask for an alphanumeric field must be left blank.

- Editing Characters for Edited Alphanumeric Fields

  - X Characters

    Each X in the editing mask represents a character position which is replaced on a one-for-one basis with a data character from the source data.

    Alphanumeric data is left-justified within the mask, and the remaining positions are space-filled to the right.

    If alphanumeric data is too long to fit the mask, the software truncates the excess right-hand characters.

    Moving data into a destination field over which is defined only X editing characters results in a straight MOVE, for no special editing is required. Therefore, a mask for an alphanumeric field must contain at least one insertion character.

- Insertion Characters for Edited Alphanumeric Fields

  All the characters in the NCR Century Code Chart except the space ( ) and X are valid insertion characters.

  - B Characters

    Each B in the editing mask separates the data edited before it from the data edited after it with a blank. The blank does not replace a data character but merely separates two adjacent data characters.

    For instance, memory contains a field of 15 characters for a store manager's first initial and his last name. To obtain a readable printout of this name, a programmer may edit the field.

Mask	Data	Output
XBXXXXXXXXXXXXXX	JBABCOCK	J BABCOCK
XBXXXXXXXXXXXXXX	PHOFFMEYER	P HOFFMEYER
XBXXXXXXXXXXXXXX	SPETERSON	S PETERSON

  - Other Characters

    Each insertion character in a mask for an alphanumeric field except the B is not replaced by a data character but, instead, is directly inserted into the destination field.

Mask	Data	Output
XX/XX/XX	021768	02/17/68

Masks for Edited Numeric Fields

To edit numeric data, move the data into the destination field over which is defined a mask. A mask for a numeric field is made up of editing characters and insertion characters. The DP positions for the definition of a mask for numeric fields must contain an entry.

- Editing Characters for Edited Numeric Fields

  - X Characters

    Each X in the editing mask represents a character position which will be replaced on a one-for-one basis with a data character from the source data.

    Numeric data is aligned on the decimal point in the mask. The edited output is zero-filled to the left and/or the right of the decimal point if the mask is longer than the source. (See the description of the decimal point insertion character for an example of zero-fill.)

    If numeric data is too long to fit the mask, the software first aligns the decimal points and then ignores the overflow to the left and/or the right of the decimal point.

    Moving data into a destination field over which is defined only X editing characters results in a straight MOVE, for no special editing is required. Therefore, a mask for a numeric field must contain at least one editing character other than X or at least one insertion character.

  - Z Characters

    Each Z in the editing mask is replaced on a one-for-one basis with a data character from the source. However, if the data character is a leading zero, the Z suppresses it.

    If a Z character is used, it must not have any editing character to the left of it except another Z or a sign (+ or -). However, the Z may have insertion characters to the left of it.

    If the mask is comprised of only Z characters and if the source value is zero, the edited output is space-filled. Therefore, to be certain of a visual output, make at least the rightmost character within the mask an X.

Mask	Data	Output
ZZZXX	00042	∅∅∅42
ZZZXX	00000	∅∅∅00
+ZZZXX	00760+	+∅∅760

- * Characters

  Each * in the editing mask is replaced on a one-for-one basis with a data character from the source. However, if the data character is a leading zero, it is replaced with an *.

  If an * character is used, it must not have any editing character to the left of it except another * or a sign (+ or -). However, it may have insertion characters to the left of it.

  If the mask is comprised only of * characters and if the source value is zero, the edited output will be asterisk-filled.

Mask	Data	Output
****XXX	0004672	***4672
****XXX	0000063	****063
****.**	000000	****.**

- + or - Characters

  A plus (+) or a minus (-) sign can appear in either the first or the last character position of the editing mask. It may not be embedded within the mask.

  A plus (+) or a minus (-) sign cannot be used in the same mask as a CR or a DB sign (explained on the next page).

  When a plus sign appears in the mask, a plus sign is output if the value of the data is positive, and a minus sign is output if the value of the data is negative.

  When a minus sign appears in the mask, a minus sign is output if the value of the data is negative. Howevever, if the value is positive, the sign is replaced with a space.

Mask	Data	Output
+XXXXX	74926+	+74926
+XXXXX	74926-	-74926
-XXXXX	74926+	☑74926
XXXXX-	74926-	74926-

- CR or DB Character Configurations

A credit (CR) or a debit (DB) sign can appear only in the two last character positions of the editing mask.

A CR or a DB sign cannot be used in the same mask as a plus (+) or a minus (-) sign.

When a CR or a DB appears in the mask, the CR or DB is output if the value of the data is negative. However, if the value is positive, the CR or DB positions are space-filled.

Mask	Data	Output
$XX.XXCR (DP 02)	2592- (DP 02)	$25.92CR
$XX.XXCR (DP 02)	2592+ (DP 02)	$25.92
$XX.XXDB (DP 02)	2592- (DP 02)	$25.92DB
$XX.XXDB (DP 02)	2592+ (DP 02)	$25.92

A mask containing the CR or DB editing characters generally also contains the decimal point and the currency symbol. (See the following discussion of Insertion Characters.)

- Insertion Characters for Edited Numeric Fields

All the characters in the Century Code Chart except the space (⍽), X, Z, *, +, -, CR, and DB are valid insertion characters.

- B Characters

Each B in the editing mask separates the data edited before it from the data edited after it with a blank. The blank does not replace a data character but merely separates two adjacent data characters.

If a programmer is to print amounts on a pre-printed form, he may elect to insert a blank character in the position taken by the pre-printed line which separates the dollar from the cent column.

Mask	Data	Output
		Amount
$XXXBXX (DP 02)	03498 (DP 02)	$034 98
$XXXBXX (DP 02)	86390 (DP 02)	$863 90

- Other Characters

Each insertion character in a mask for a numeric field except the B is not replaced by a data character but, instead, is directly inserted into the destination field. However, if the character preceding the insertion character is suppressed, the insertion character (except the decimal point) is also suppressed.

Mask	Data	Output
$ZZZ,ZZX.XX (DP 02)	0099 (DP 02)	$ 0.99
$ZZZ.ZZ (DP 02)	0009 (DP 02)	$ .09
$ZZZ.ZZ (DP 02)	0000 (DP 02)	⍽⍽⍽⍽⍽⍽⍽
$ZZZ.XX (DP 02)	0000 (DP 02)	$ .00

Certain insertion characters are commonly found in masks for numeric fields. These characters -- the currency symbol and the decimal point -- have special rules governing their use.

- . or , Characters

Usually, either the period or the comma is used to designate the decimal point; however, any character except the editing symbols (+, -, B, *, Z, X and floating $) may be used to designate the decimal point. Software first aligns the decimal point in the mask with the decimal point in the source data and then edits the source data.

Mask	Data	Output
$XXX.XX (DP 02)	19476 (DP 02)	$194.76
£X.XXX,XX (DP 02)	519476 (DP 02)	£5.194,76

To determine which character is the decimal point, the NEAT/3 language uses the DP entry of the source statement defining the mask.

Mask	Data	Output
$XXXDXX (DP 02)	19476 (DP 02)	$194D76

However, if desired, the decimal-point position may not be designated but merely implied by the DP entry of the mask.

Mask	Data	Output
$XXXXXX (DP 02)	194760 (DP 02)	$194760
XXXXXX (DP 03)	69219 (DP 02)	692190

If the number of integer positions or of decimal positions in the source data is too long to fit the mask, the software truncates the integer or the decimal overflow. Therefore, the programmer must exercise care when defining the length of the mask.

Mask	Data	Output
XXX.XXXX (DP 04)	3765913 (DP 05)	037.6591
XXX.XXXX (DP 04)	3765913 (DP 03)	765.9130

A decimal point symbol (however it may be designated) is not suppressed unless the complete field is suppressed.

Mask	Data	Output
ZZZ.ZZ (DP 02)	00000 (DP 02)	spaces
ZZZ,ZZ (DP 02)	00001 (DP 02)	⍁⍁⍁,01

- $ or £ Characters

A single currency symbol in the mask is treated as an insertion character. Any insertion character or editing character may appear to the left of the currency symbol.

Mask		Data		Output
£XX,XX	(DP 02)	4900	(DP 02)	£49,00
$XXXXXXX	(DP 02)	19498	(DP 02)	$0019498
-$XX.XX	(DP 02)	9658+	(DP 02)	☑$96.58

- $$ or ££ Character Configurations

The floating currency symbol, a variation of the regular currency symbol, is a series of at least two successive currency symbols. Only insertion characters or a sign (+ or -) may appear to the left of the floating-currency-symbol configuration.

The processor first aligns the decimal point in the mask with the decimal point in the source data and then edits the source.

Each character in the floating-currency-symbol configuration is individually analyzed.

- If the corresponding data character is a significant data character, the data is output.

- If the corresponding data character is a leading zero, the currency symbol corresponding to the rightmost leading zero is output, and the currency symbols to the left of this currency symbol are suppressed.

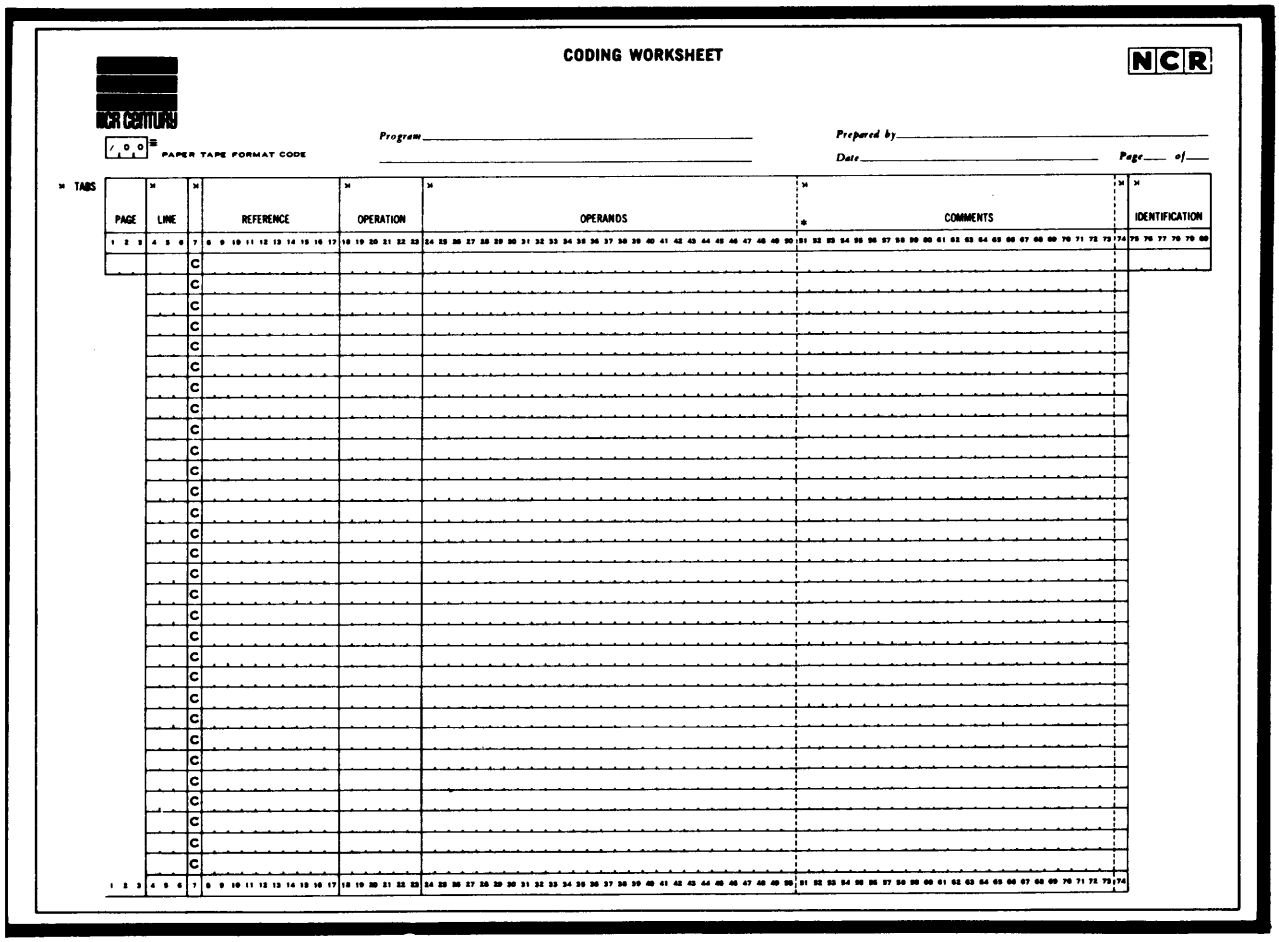
Mask		Data		Output
$$$XXX.XX	(DP 02)	402700	(DP 02)	☑$4027.00
£££XX,XX	(DP 02)	9645	(DP 02)	☑☑☑£96,45
-$$.XX	(DP 02)	143-	(DP 02)	-$1.43
$$$.XX	(DP 02)	00001	(DP 02)	☑☑ $.01
$$$.$$	(DP 02)	00000	(DP 02)	☑☑☑☑☑☑

* * * *

CODING SHEETS

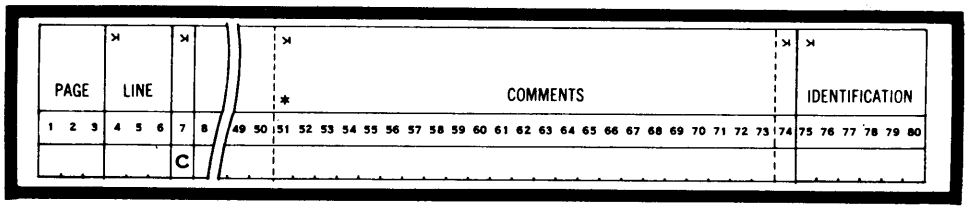
ENTRIES ON CODING SHEETS

The coding sheets for the NEAT/3 Compiler are those forms upon which the programmer codes his procedural instructions. A coding sheet is illustrated. Note that a maximum of 30 instructions can be coded on each sheet, one instruction for each line.



Common Entries

The header, the paper tape format code, the page-and-line number (positions 1-6), the worksheet code (position 7), the comments, the delete digit (position 74), and the identification tag (positions 75-80) are defined in INTRODUCTION AND DATA, tab 3, "Programming Worksheets".



Reference

- Symbolic Reference Tag

  A programmer may enter a symbolic reference tag on coding sheets in positions 8-17 to identify the string of instructions being coded.

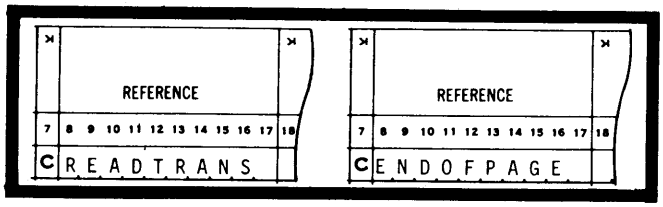
  Usually, the programmer enters a near-English word which allows him immediately to identify the instruction string to which it refers, e.g. ENDOFPAGE . These reference tags become entrances to sections of coding that may be executed many times during processing. For instance, after a program reads a transaction record, finds its master record, and updates the master record, it branches to reexecute the same coding.

  The reference tag may contain from 1 to 10 characters which are made up of the alphabet (A-Z) and/or the numerals (0-9). Each tag must begin in position 8 and must contain at least one alphabetical character. Spaces are not permitted within the tag.

  Unless it is used with a qualifier (see the discussion of operands in this publication), each tag must be unique to the program; i.e., it may appear as a reference tag in only one source statement in the program. However, the name may appear as an operand as often as necessary.

  The compiler, as it processes each source statement, checks positions 8-17 for an entry. If it finds a reference tag, the compiler associates with this reference tag all information pertinent to where the instruction is stored.

  The following are examples of correctly entered reference tags:



- Local Tag

A programmer may enter a local tag on coding sheets in positions 8-10 to identify a statement within a program region. (A program region begins at a source line with a symbolic reference tag and continues up to but not including the next source line with a symbolic reference tag.)

A local tag may be used as an operand only within the program region in which the tag appears. However, a local tag may not be used in a qualified operand. (See Operands in this publication).

Since local tags are not included in the compiler cross-reference listing, they are used to rid the listing of symbolic reference tags of minor significance. Also, if local tags are used instead of symbolic reference tags, compilation time is reduced.

A local tag contains three characters; the first must be a dollar sign, and the remaining two must be within the range from 00 to 24. Local tags need be unique only within the particular program region in which they appear. For example, two or more program regions in the same program may each contain a local tag $09; however, $09 can appear only once within each region.

In the following example, the first program region contains one local tag, $09. The second program region contains two local tags: $08 and $09.

	LINE		REFERENCE	OPERATION	
	4 5 6	7	8 9 10 11 12 13 14 15 16 17	18 19 20 21 22 23	24 25 26 27 28 29 30
First Program Region	030	C	LESS	GET	
	060	C	$09	ADD	
	090	C		COMP	
	120	C		BRL	$09
Second Program Region	150	C	NOTLESS	BRE	$09
	180	C	$08	SUB	
	210	C		COMP	
	240	C		BRG	$08
	270	C	$09	MOVE	

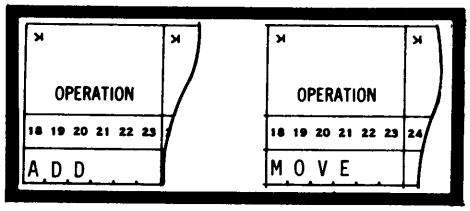
Note that the procedural instructions at both lines 120 and 150 transfer control to the local tag $09. However, since the instructions are in different program regions, the instruction at line 120 (branch if less) transfers control to line 060, and the instruction at line 150 (branch if equal) transfers control to line 270.

Operation

Beginning in position 18, enter the name of the instruction to be performed. Two types of coding instructions may be entered: procedural instructions and compiler control instructions.

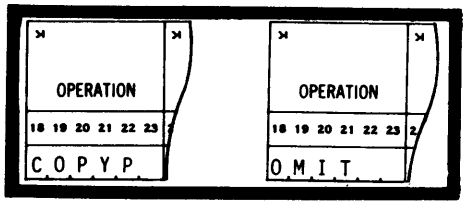
Procedural instructions make up the logic flow of the program. They tell the processor to add, subtract, move data to another location, print a record, store data on disc, etc. The compiler translates these instructions into machine language.

The following examples show correctly entered procedural instructions:



Compiler control instructions may also be entered on a coding sheet. These instructions direct the compiler to perform special tasks. These instructions do not become part of the object program, but they do affect the structure of the object program.

The following examples show correctly entered compiler control instructions:

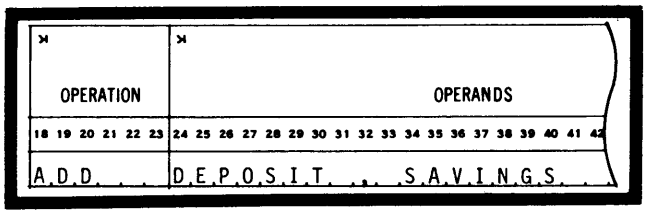


Operands

Most procedural instructions require the programmer to enter at least one operand in the source statement. The procedural instruction tells what is to be done; the operand tells upon which data or quantity the operation is to be performed or to which instruction control is to be transferred.

For example, in the instruction: ADD the contents of DEPOSIT to the contents of SAVINGS, ADD is the procedural instruction, and both DEPOSIT and SAVINGS are the operands. Likewise, in the instruction: branch to the source line referenced by READMASTER, branch is the procedural instruction, and READMASTER is the operand.

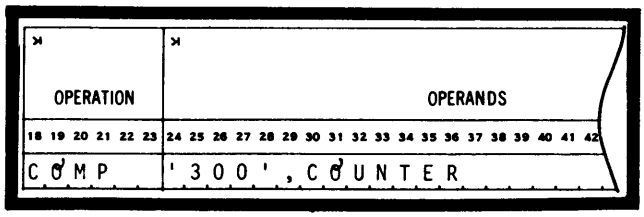
When two or more operands are used, separate them with a comma. Spaces may appear on either side of the comma to make the statement more readable. Consider the following example:



There are three types of operands: literal, reference, and qualified. Let's consider each of these in detail.

- Literal Operand

  A literal operand is a constant that represents actual data. For example, in the instruction illustrated below, the actual value of 300 is compared to the contents of COUNTER.



  A literal can be either numeric or alphanumeric. A numeric literal may be composed of the characters 0-9, and the plus, minus, and period (decimal point) symbols. (If a numeric literal contains either a plus or minus sign, it must be in the leftmost position.) An alphanumeric literal may be composed of any of the USASI characters except the single quote. Every literal operand must be enclosed in single quotation marks, e.g. '24', 'TOTAL'.

  NOTE

  Write single quotation marks legibly so that they can be easily distinguished by the keypunch operator.

  When a literal operand is used with another operand in a compare, move, or arithmetic operation, the compiler assigns to the literal the same data type and length as the associated operand's. For example, if COUNTER (illustrated above) is an alphanumeric (X-type) field with a length of 5, the '300' is assigned an X-type format of 300☐☐. If COUNTER is an unsigned decimal (U-type) field with a length of 5, the '300' is assigned a U-type format of 00300. (For further information concerning data field formats, refer to INSTRUCTIONS, tab 2, "MOVE Instruction" (Pub. No. 4) in this manual.)

- Reference Operand

A reference operand may be a symbolic reference tag of a data definition, or it may be either a symbolic reference tag or a local tag of a procedural instruction.

If the operand is a symbolic reference tag of a data definition, the operation is performed upon the data whether the data be a file, record, field, area, or table. In the following example, the key of the current transaction record is compared to the key of the current master record.

```
OPERATION      OPERANDS
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
C O M P .      T R A N S K E Y . , . M A S T E R K E Y .
```

When the operand is a reference tag (either a symbolic or local) of an instruction, control is transferred to this instruction if the condition set forth by the operation holds true. In the following example, control is transferred to READMASTER if the contents of TRANSKEY is greater than the contents of MASTERKEY.

```
OPERATION      OPERANDS
18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
C O M P .      T R A N S K E Y . , . M A S T E R K E Y .
B R G .        R E A D M A S T E R .
```

** - Qualified Operand

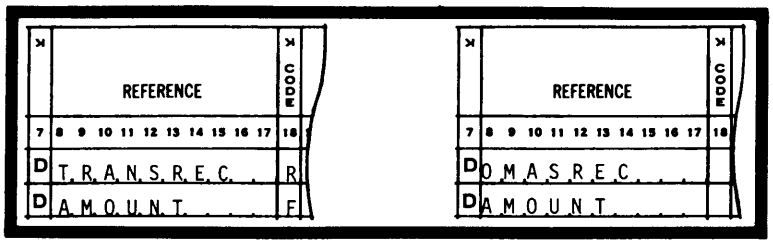
A qualified operand specifies which of two or more identical reference tags is desired.

The qualified operand consists of a composite name of two or more symbolic reference tags that are separated by a period. Each reference tag within the operand must be a legal symbolic reference tag or a data name of either a procedural instruction or a data unit (whether the data be a file, record, field, area, or table).

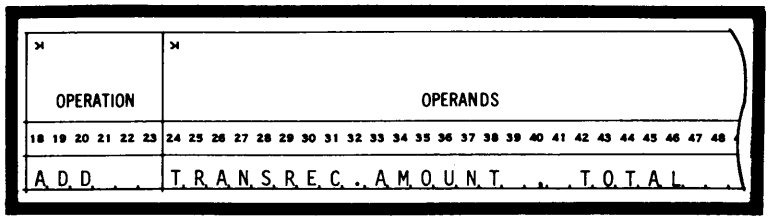
The first tag, the qualifier, must specify a unique data name or symbolic reference tag in the program. The second tag, separated from the qualifier by a period, need not be unique to the program. The only restriction to the second tag is that it must be contained in the same section or overlay as is the qualifier.

In searching for the location of the data or instruction to be operated upon, the compiler first finds the location of the first tag (which is unique to the program). It then searches the source statements immediately following the location of this qualifier until it finds the first occurrence of the second tag. The data or instruction at this second location is that which will be acted upon by the procedural instruction during the production run of the program.

For example, a transaction record (TRANSREC) may contain a field referenced by AMOUNT, and an old master record (OMASREC) may also contain a field referenced by AMOUNT. Consider the following excerpts of record and field definitions:



The programmer, wishing to manipulate the data, must specify which AMOUNT he wants, i.e. either TRANSREC.AMOUNT or OMASREC.AMOUNT. He specifies this in the operands positions of the instruction that manipulates this field. For instance, the following instruction adds the contents of AMOUNT in the transaction record to the contents of the field with a unique reference tag TOTAL:



The AMOUNT field in the old master record is not affected by the above instruction.

- Conventions for Using Qualified Operands

  NEAT/3 qualification is based on source program presentation sequence; that is, each successive symbol in a qualified series must physically follow the previous symbol in the series in the source program. For example, in the qualified series A.B.C., the symbol C must follow the symbol B and the symbol B must follow the symbol A in the source program; however, the symbols need not be contiguous in the source program.

  For qualification purposes, the NEAT/3 language is divided into four levels: the section level, the file level, the area level, and the coding level. The first element (qualifier) of a qualified series determines which level is applicable. Once a level is determined, qualification must stay within that

level (although a program may contain several files, qualification may function only within one file level).

Special qualification rules apply to the OVRLAY and ENTRY statements. The reference of an ENTRY statement may only be qualified by the reference of an OVRLAY statement or by the reference of another ENTRY statement.

- Section Level

  This level, which corresponds directly with the program section, includes everything within the program section. This level is entered if the qualifier of a qualified series is the reference of a SECT instruction. Subsequent elements in the series may refer to any reference within the section.

  This level extends from a SECT instruction to the next SECT or OVRLAY instruction, or END$.

- File Level

  This level which corresponds directly with a program file definition, includes all record, field, table, and item definitions within the file. This level is entered if the qualifier of a qualified series is a reference within a file.

  This level extends from a file specification worksheet to the next file specification worksheet, the next area statement, the next coding statement, or to the next SECT or OVRLAY instruction, or END$.

- Area Level

  This level, which corresponds directly with a program area definition, includes all fields, table, and item definitions within the area. This level is entered if the qualifier of a qualified series is a reference within the area.

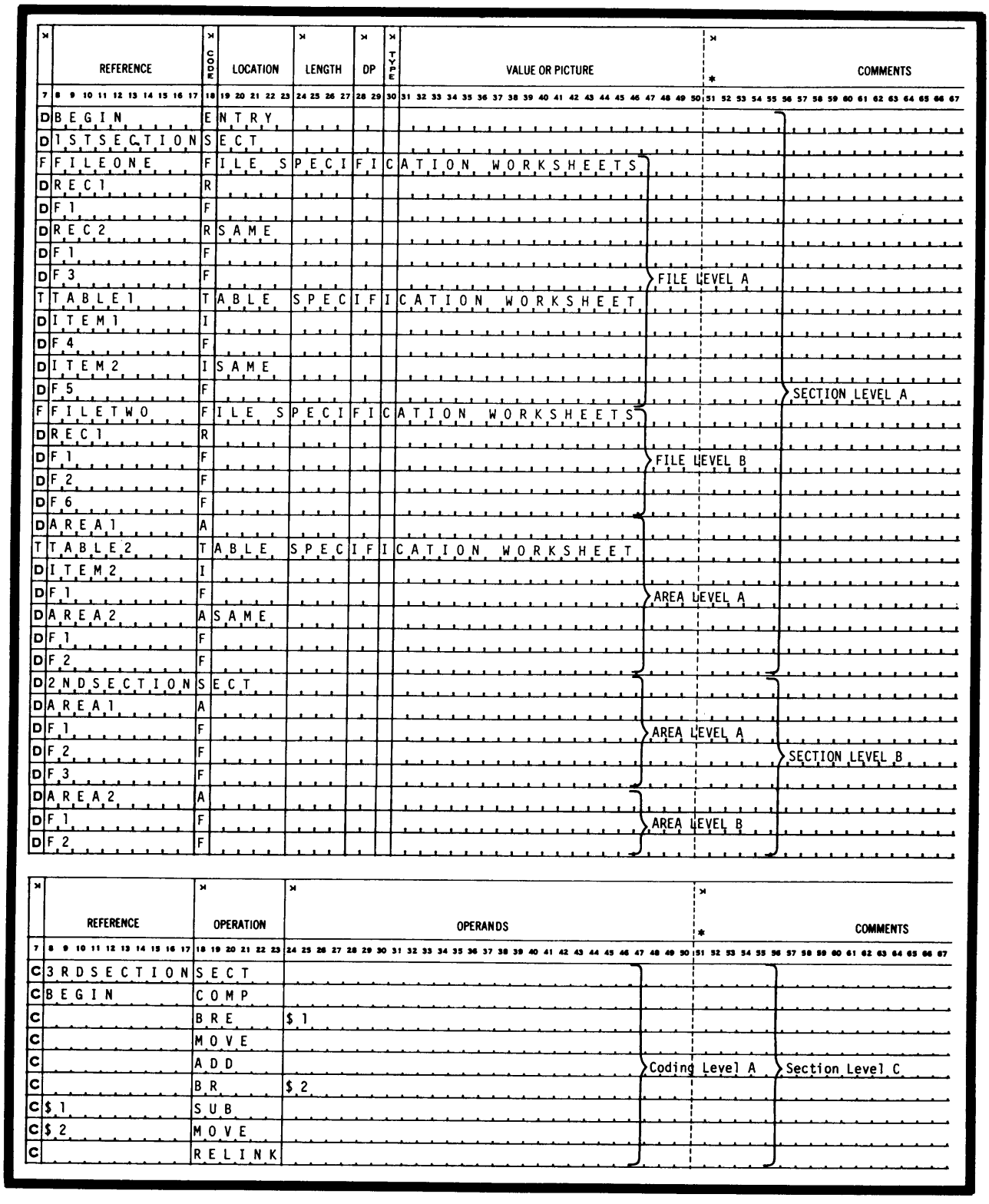
  This level extends from an area statement to either the next non-SAME area statement, or to the next SECT or OVRLAY instruction or END$.

- Coding Level

  This level, which corresponds directly with the program coding statements is entered if the qualifier of a qualified series is the reference of a coding statement.

  This level extends from the first coding statement to the next SECT or OVRLAY instruction, or END$.

  Consider the following illustration and explanation.


REFERENCE
CODE
LOCATION
LENGTH
DP
TYPE
VALUE OR PICTURE
COMMENTS
BEGIN ENTRY
1STSECTION SECT
FFILEONE FILE SPECIFICATION WORKSHEETS
REC1 R
F1 F
REC2 R SAME
F1 F
F3 F
TABLE1 T ABLE SPECIFICATION WORKSHEET
ITEM1 I
F4 F
ITEM2 I SAME
F5 F
FFILETWO F ILE SPECIFICATION WORKSHEETS
REC1 R
F1 F
F2 F
F6 F
AREA1 A
TTABLE2 T ABLE SPECIFICATION WORKSHEET
ITEM2 I
F1 F
AREA2 A SAME
F1 F
F2 F
2NDSECTION SECT
AREA1 A
F1 F
F2 F
F3 F
AREA2 A
F1 F
F2 F
FILE LEVEL A
SECTION LEVEL A
FILE LEVEL B
AREA LEVEL A
AREA LEVEL A
SECTION LEVEL B
AREA LEVEL B
REFERENCE
OPERATION
OPERANDS
COMMENTS
3RDSECTION SECT
BEGIN COMP
BRE $1
MOVE
ADD
BR $2
$1 SUB
$2 MOVE
RELINK
Coding Level A
Section Level C

Following are examples of legal qualified series from the illustration.

```
FILEONE.REC1
FILEONE.REC1.F.
FILEONE.REC2.F1
FILEONE.REC1.F3  (The compiler will not comment on this qualified series
                  even though it spans two record description.)
REC2.F1
TABLE1.ITEM1
TABLE1.ITEM1.F4
TABLE2.ITEM1.F5  (The compiler will not comment on this qualified series
                  even though it spans two item descriptions.)
2NDSECTION.AREA1
3RDSECTION.BEGIN
```

Following are examples of illegal qualified series from the illustration.

FILEONE.F6	F6 is not contained within File Level A.
REC1.F1	REC1 is not unique within Section Level A. This series should be written FILEONE.REC1.F1.
AREA2.F3	F3 is not contained within Area Level A of Section Level A.
AREA1.F3	Since this series is an operand of the coding in Section Level C, AREA1 is ambiguous. This series should be written 2NDSECTION.AREA1.F3.
FILETWO.TABLE2	TABLE2 is not contained within File Level B.
2NDSECTION.BEGIN	BEGIN is not contained within Section Level B.
3RDSECTION.$1	Local references may not be used in a qualified series.
2NDSECTION.F1.AREA1	AREA1 is incorrect since it does not physically follow F1 in the section level.

Continuation Line

As stated before, a procedural instruction may have more than one operand. The length of the operands may extend to character position 73 on the coding sheet. However, if the operands overflow this limit, they may be continued on a second source line. This second line is called a continuation line.

If a continuation line is used, the programmer must strictly adhere to these rules:

- First Line

  Fill out the source line as usual. Entries in the following positions are valid: page-and-line number, reference, operation, operands, delete digit, and identification.

  End the operand entry with a complete operand followed by a comma.

- Continuation Line

  Normal rules apply to the entries in the following positions: page-and-line number, delete digit, and identification.

  Enter a hyphen (-) in position 18. Leave the reference and remaining operation positions blank; if these positions are not blank, the compiler flags this source line as an error statement and ignores it.

  Continue entering the operands in position 24. Normal rules apply to these operands.

  A comment may begin with an asterisk in the first available position following the operands. If this comment overflows the limit (character position 73 if input is from punched cards, or character position 100 if input is from punched paper tape), it may be continued on the next line by placing an asterisk in position 8 of that next line. (See the Complete-Line Comment under INTRODUCTION AND DATA, tab 3, "Programming Worksheets".)

In the following example, the amount in the transaction record is added to the amount in the old master record, and the result is stored in the field or area referenced by TOTALAMT. If, however, the result is too large to fit the space reserved for TOTALAMT, control is transferred to a routine referenced by TOOBIG.

OPERATION	OPERANDS	COMMENTS
18 19 20 21 22 23	24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50	51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
ADDC	TRANSREC.AMOUNT , OMASREC.AMOUNT , TOTALAMT,	
-	TOOBIG	

* * * *